Spring Cloud 为开发人员提供了快速构建分布式系统中一些常见模式的工具（例如配置管理，服务发现，断路器，智能路由，微代理，控制总线）。分布式系统的协调导致了样板模式，使用 Spring Cloud 开发人员可以快速地支持实现这些模式的服务和应用程序。他们将在任何分布式环境中运行良好，包括开发人员自己的笔记本电脑，裸机数据中心，以及 Cloud Foundry 等托管平台。

版本：Dalston.RELEASE

特性

Spring Cloud 专注于提供良好的开箱即用经验的典型用例和可扩展性机制覆盖。

- 分布式/版本化配置
- 服务注册和发现
- 路由
- service - to - service 调用
- 负载均衡
- 断路器
- 分布式消息传递

云原生应用程序

VISIT…

LANZAROTE

Caliente.com

云原生是一种应用开发风格，鼓励在持续交付和价值驱动开发领域轻松采用最佳实践。相关的学科是建立 12-factor Apps，其中开发实践与交付和运营目标相一致，例如通过使用声明式编程和管理和监控。Spring Cloud 以多种具体方式促进这些开发风格，起点是一组功能，分布式系统中的所有组件都需要或需要时轻松访问。

许多这些功能都由 Spring Boot 覆盖，我们在 Spring Cloud 中建立。更多的由 Spring Cloud 提供为两个库：Spring Cloud Context 和 Spring Cloud Commons。Spring Cloud 上下文为 Spring Cloud 应用程序（引导上下文，加密，刷新范围和环境端点）的 `ApplicationContext` 提供实用程序和特殊服务。Spring Cloud Commons 是一组在不同的 Spring Cloud 实现中使用的抽象和常用类（例如 Spring Cloud Netflix vs. Spring Cloud Consul）。

如果由于“非法密钥大小”而导致异常，并且您正在使用 Sun 的 JDK，则需要安装 Java 加密扩展（JCE）无限强度管理策略文件。有关详细信息，请参阅以下链接：

- Java 6 JCE
- Java 7 JCE
- Java 8 JCE

将文件解压缩到 JDK / jre / lib / security 文件夹（无论您使用的是哪个版本的 JRE / JDK x64 / x86）。

注意	Spring Cloud 根据非限制性 Apache 2.0 许可证发布。如果您想为文档的这一部分做 中找到到项目中的源代码和问题跟踪器。

Spring Cloud 上下文：应用程序上下文服务

Spring Boot 对于如何使用 Spring 构建应用程序有一个看法：例如它具有常规配置文件的常规位置，以及用于常见管理和监视任务的端点。Spring Cloud 建立在此之上，并添加了一些可能系统中所有组件将使用或偶尔需要的功能。

引导应用程序上下文

一个 Spring Cloud 应用程序通过创建一个“引导”上下文来进行操作，这个上下文是主应用程序的父上下文。开箱即用，负责从外部源加载配置属性，还解密本地外部配置文件中的属性。这两个上下文共享一个 `Environment`，这是任何 Spring 应用程序的外部属性的来源。Bootstrap 属性的优先级高，因此默认情况下不能被本地配置覆盖。

引导上下文使用与主应用程序上下文不同的外部配置约定，因此使用 `bootstrap.yml` `application.yml`（或`.properties`）代替引导和主上下文的外部配置。例：

bootstrap.yml

```
spring:
  application:
    name: foo
  cloud:
    config:
      uri: ${SPRING_CONFIG_URI:http://localhost:8888}
```

如果您的应用程序需要服务器上的特定于应用程序的配置，那么设置 `spring.application.name`（在 `bootstrap.yml` 或 `application.yml`）中是个好主意。

您可以通过设置 `spring.cloud.bootstrap.enabled=false`（例如在系统属性中）来完全禁用引导过程。

应用程序上下文层次结构

如果您从 `SpringApplication` 或 `SpringApplicationBuilder` 构建应用程序上下文，则将 Bootstrap 上下文添加为该上下文的父级。这是一个 Spring 的功能，即子上下文从其父进程继承属性源和配置文件，因此与不使用 Spring Cloud Config 构建相同上下文相比，“主”应用程序上下文将包含其他属性源。额外的财产来源是：

- “bootstrap”：如果在 Bootstrap 上下文中找到任何 `PropertySourceLocators`，则可选 `CompositePropertySource` 显示为高优先级，并且具有非空属性。一个例子是来自 Spring Cloud Config 服务器的属性。有关如何自定义此属性源的内容的说明，请参阅 下文。
- “applicationConfig：[classpath：bootstrap.yml]”（如果 Spring 配置文件处于活动状态，则为朋友）。如果您有一个 `bootstrap.yml`（或属性），那么这些属性用于配置引导上下文，然后在父进程设置时将它们添加到子上下文中。它们的优先级低于 `application.yml`（或属性）以及作为创建 Spring

Boot 应用程序的过程的正常部分添加到子级的任何其他属性源。有关如何自定义这些属性源的内容的说明，请参阅下文。

由于属性源的排序规则，“引导”条目优先，但请注意，这些条目不包含来自 `bootstrap.yml` 的任何数据，它具有非常低的优先级，但可用于设置默认值。

您可以通过简单地设置您创建的任何 `ApplicationContext` 的父上下文来扩展上下文层次结构，例如使用自己的界面，或使用 `SpringApplicationBuilder` 方便方法（`parent()`，`child()` 和 `sibling()`）。引导环境将是您创建自己的最高级祖先的父级。层次结构中的每个上下文都将有自己的“引导”属性源（可能为空），以避免无意中将值从父级升级到其后代。层次结构中的每个上下文（原则上）也可以具有不同的 `spring.application.name`，因此如果存在配置服务器，则不同的远程属性源。普通的 Spring 应用程序上下文行为规则适用于属性解析：子环境中的属性通过名称和属性源名称覆盖父项中的属性（如果子级具有与父级名称相同的属性源，一个来自父母的孩子不包括在孩子中）。

请注意，`SpringApplicationBuilder` 允许您在整个层次结构中共享 `Environment`，但这不是默认值。因此，兄弟情境尤其不需要具有相同的资料或财产来源，尽管它们与父母共享共同点。

改变引导位置 Properties

可以使用 `spring.cloud.bootstrap.name`（默认“引导”）或 `spring.cloud.bootstrap.location`（默认为空）指定 `bootstrap.yml`（或`.properties`）位置，例如在系统属性中。这些属性的行为类似于具有相同名称的 `spring.config.*`变体，实际上它们用于通过在其 `Environment` 中设置这些属性来设置引导 `ApplicationContext`。如果在正在构建的上下文中有活动的配置文件（来自 `spring.profiles.active` 或通过 `Environment` API））,则该配置文件中的属性也将被加载，就像常规的 Spring Boot 应用程序，例如来自 `bootstrap-development.properties` 的“开发”简介。

覆盖远程 Properties 的值

通过引导上下文添加到应用程序的属性源通常是“远程”（例如从配置服务器），并且默认情况下，不能在本地覆盖，除了在命令行上。如果要允许您的应用程序使用自己的系统属性或配置文件覆盖远程属性，则远程属性源必须通过设置 `spring.cloud.config.allowOverride=true`（在本地设置本身不起作用）授予权限。一旦设置了该标志，就会有一些更精细的设置来控制远程属性与系统属性和应用程序本地配置的位置：`spring.cloud.config.overrideNone=true` 覆盖任何本地属性源，`spring.cloud.config.overrideSystemProperties=false` 如果只有系统属性和 env var 应该覆盖远程设置，而不是本地配置文件。

自定义引导配置

可以通过在 `org.springframework.cloud.bootstrap.BootstrapConfiguration` 键下添加条目`/META-INF/spring.factories` 来训练引导上下文来执行任何您喜欢的操作。这是用于创建上下文的 Spring `@Configuration` 类的逗号分隔列表。您可以在此处创建要用于自动装配的主应用程序上下文的任何 bean，并且还有 `ApplicationContextInitializer` 类型的`@Beans` 的特殊合同。如果要控制启动顺序（默认顺序为"最后"），可以使用`@Order` 标记类。

警告	添加自定义 `BootstrapConfiguration` 时，请注意，您添加的类不是错误的`@Co` 序上下文中，可能不需要它们。对于您的`@ComponentScan` 或`@SpringBootAppl` 置类，请使用单独的包名称。

引导过程通过将初始化器注入主 `SpringApplication` 实例（即正常的 Spring Boot 启动顺序，无论是作为独立应用程序运行还是部署在应用程序服务器中）结束。首先，从 `spring.factories` 中找到的类创建引导上下文，然后在 `ApplicationContextInitializer` 类型的所有`@Beans` 添加到主 `SpringApplication` 开始之前。

自定义引导属性源

引导过程添加的外部配置的默认属性源是 Config Server，但您可以通过将 `PropertySourceLocator` 类型的 bean 添加到引导上下文（通过 `spring.factories`）添加其他源。您可以使用此方法从其他服务器或数据库中插入其他属性。

作为一个例子，请考虑以下微不足道的自定义定位器：

```
@Configuration
public class CustomPropertySourceLocator implements
PropertySourceLocator {

    @Override
    public PropertySource<?> locate(Environment
environment) {
        return new MapPropertySource("customProperty",
                Collections.<String,
Object>singletonMap("property.from.sample.custom.source",
"worked as intended"));
    }

}
```

传入的 `Environment` 是要创建的 `ApplicationContext` 的 `Environment`，即为我们提供额外的属性来源的。它将已经具有正常的 Spring Boot 提供的资源来源，因此您可以使用它们来定位特定于此 `Environment` 的属性源（例如通过将其绑定在 `spring.application.name` 上，如在默认情况下所做的那样 Config Server 属性源定位器）。

如果你在这个类中创建一个 jar，然后添加一个 `META-INF/spring.factories` 包含：

```
org.springframework.cloud.bootstrap.BootstrapConfiguratio
n=sample.custom.CustomPropertySourceLocator
```

那么“customProperty”`PropertySource` 将显示在其类路径中包含该 jar 的任何应用程序中。

环境变化

应用程序将收听`EnvironmentChangeEvent`，并以几种标准方式进行更改（用户可以以常规方式添加`ApplicationListeners`附加`ApplicationListeners`）。当观察到`EnvironmentChangeEvent`时，它将有一个已更改的键值列表，应用程序将使用以下内容：

- 重新绑定上下文中的任何`@ConfigurationProperties` bean
- 为`logging.level.*`中的任何属性设置记录器级别

请注意，配置客户端不会通过默认轮询查找`Environment`中的更改，通常我们不建议检测更改的方法（尽管可以使用`@Scheduled`注释进行设置）。如果您有一个扩展的客户端应用程序，那么最好将`EnvironmentChangeEvent`广播到所有实例，而不是让它们轮询更改（例如使用 Spring Cloud 总线）。

`EnvironmentChangeEvent`涵盖了大量的刷新用例，只要您真的可以更改`Environment`并发布事件（这些 API 是公开的，部分内核为 Spring）。您可以通过访问`/configprops`端点（普通 Spring Boot 执行器功能）来验证更改是否绑定到`@ConfigurationProperties` bean。例如，`DataSource`可以在运行时更改其`maxPoolSize`（由 Spring Boot 创建的默认`DataSource`是一个`@ConfigurationProperties` bean），并且动态增加容量。重新绑定`@ConfigurationProperties`不会覆盖另一大类用例，您需要更多的控制刷新，并且您需要更改在整个`ApplicationContext`上是原子的。为了解决这些担忧，我们有`@RefreshScope`。

刷新范围

当配置更改时，标有`@RefreshScope`的 Spring `@Bean` 将得到特殊处理。这解决了状态 bean 在初始化时只注入配置的问题。例如，如果通过 `Environment` 更改数据库 URL 时 `DataSource` 有开放连接，那么我们可能希望这些连接的持有人能够完成他们正在做的工作。然后下一次有人从游泳池借用一个连接，他得到一个新的 URL。

刷新范围 bean 是在使用时初始化的懒惰代理（即当调用一个方法时），并且作用域作为初始值的缓存。要强制 bean 重新初始化下一个方法调用，您只需要使其缓存条目无效。

`RefreshScope` 是上下文中的一个 bean，它有一个公共方法 `refreshAll()`来清除目标缓存中的范围内的所有 bean。还有一个 `refresh(String)`方法可以按名称刷新单个 bean。此功能在`/refresh` 端点（通过 HTTP 或 JMX）中公开。

注意	`@RefreshScope`（技术上）在`@Configuration` 类上工作，但可能会导致令人惊讶中定义的所有有`@Beans` 本身都是`@RefreshScope`。具体来说，任何取决于这些 bean 被更新，除非它本身在`@RefreshScope`（在其中将重新刷新并重新注入其依赖关系 `@Configuration`）重新初始化。

加密和解密

Spring Cloud 具有一个用于在本地解密属性值的 `Environment` 预处理器。它遵循与 Config Server 相同的规则，并通过 `encrypt.*`具有相同的外部配置。因此，您可以使用`{cipher}*`格式的加密值，只要有一个有效的密钥，那么在主应用程序上下文获取 `Environment` 之前，它们将被解密。要在应用程序中使用加密功能，您需要在您的类路径中包含 Spring 安全性 RSA（Maven 协调

“org.springframework.security:spring-security-rsa”），并且还需要全面强大的 JCE 扩展你的 JVM

如果由于“非法密钥大小”而导致异常，并且您正在使用 Sun 的 JDK，则需要安装 Java 加密扩展（JCE）无限强度管理策略文件。有关详细信息，请参阅以下链接：

- Java 6 JCE
- Java 7 JCE
- Java 8 JCE

将文件解压缩到 JDK / jre / lib / security 文件夹（无论您使用的是哪个版本的 JRE / JDK x64 / x86）。

端点

对于 Spring Boot 执行器应用程序，还有一些额外的管理端点：

- POST 到`/env`以更新`Environment`并重新绑定`@ConfigurationProperties`和日志级别
- `/refresh`重新加载引导带上下文并刷新`@RefreshScope` bean
- `/restart`关闭`ApplicationContext`并重新启动（默认情况下禁用）
- `/pause`和`/resume`调用`Lifecycle`方法（`stop()`和`start()` `ApplicationContext`）

Spring Cloud Commons：普通抽象

诸如服务发现，负载平衡和断路器之类的模式适用于所有 Spring Cloud 客户端可以独立于实现（例如通过 Eureka 或 Consul 发现）的消耗的共同抽象层。

@EnableDiscoveryClient

Commons 提供`@EnableDiscoveryClient`注释。这通过 `META-INF/spring.factories` 查找 `DiscoveryClient` 接口的实现。Discovery Client 的实现将在 `org.springframework.cloud.client.discovery.EnableDiscoveryClient` 键下的 `spring.factories` 中添加一个配置类。`DiscoveryClient` 实现的示例是 Spring Cloud Netflix Eureka，Spring Cloud Consul 发现和 Spring Cloud Zookeeper 发现。

默认情况下，`DiscoveryClient` 的实现将使用远程发现服务器自动注册本地 Spring Boot 服务器。可以通过在`@EnableDiscoveryClient` 中设置 `autoRegister=false` 来禁用此功能。

ServiceRegistry

Commons 现在提供了一个 `ServiceRegistry` 接口，它提供了诸如 `register(Registration)` 和 `deregister(Registration)` 之类的方法，允许您提供定制的注册服务。`Registration` 是一个标记界面。

```
@Configuration
@EnableDiscoveryClient(autoRegister=false)
public class MyConfiguration {
    private ServiceRegistry registry;

    public MyConfiguration(ServiceRegistry registry) {
        this.registry = registry;
    }

    // called via some external process, such as an event
or a custom actuator endpoint
    public void register() {
        Registration registration =
constructRegistration();
        this.registry.register(registration);
    }
}
```

每个 `ServiceRegistry` 实现都有自己的 `Registry` 实现。

服务部门自动注册

默认情况下，`ServiceRegistry` 实现将自动注册正在运行的服务。要禁用该行为，有两种方法。您可以设置 `@EnableDiscoveryClient(autoRegister=false)` 永久禁用自动注册。您还可以设置 `spring.cloud.service-registry.auto-registration.enabled=false` 以通过配置禁用该行为。

服务注册执行器端点

Commons 提供`/service-registry`致动器端点。该端点依赖于 Spring 应用程序上下文中的 `Registration` bean。通过 GET 调用`/service-registry/instance-status` 将返回 `Registration` 的状态。具有 `String` 主体的同一端点的 POST 将将当前 `Registration` 的状态更改为新值。请参阅您正在使用的 `ServiceRegistry` 实现的文档，以获取更新状态的允许值和为状态获取的值。

Spring RestTemplate 作为负载平衡器客户端

`RestTemplate` 可以自动配置为使用功能区。要创建负载平衡 `RestTemplate` 创建 `RestTemplate` `@Bean` 并使用`@LoadBalanced` 限定符。

警告	通过自动配置不再创建 `RestTemplate` bean。它必须由单个应用程序创建。

```
@Configuration
public class MyConfiguration {

    @LoadBalanced
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

public class MyClass {
    @Autowired
    private RestTemplate restTemplate;

    public String doOtherStuff() {
        String results =
restTemplate.getForObject("http://stores/stores",
String.class);
        return results;
    }
}
```

URI 需要使用虚拟主机名（即服务名称，而不是主机名）。Ribbon 客户端用于创建完整的物理地址。有关 如何设置 RestTemplate 的详细信息，请参阅 RibbonAutoConfiguration。

重试失败的请求

负载平衡 RestTemplate 可以配置为重试失败的请求。默认情况下，该逻辑被禁用，您可以通过将 Spring 重试添加到应用程序的类路径来启用它。负载平衡 RestTemplate 将符合与重试失败请求相关的一些 Ribbon 配置值。如果要在类路径中使用 Spring 重试来禁用重试逻辑，则可以设置 spring.cloud.loadbalancer.retry.enabled=false。您可以使用的属性是 client.ribbon.MaxAutoRetries，client.ribbon.MaxAutoRetriesNextServer 和 client.ribbon.OkToRetryOnAllOperations。请参阅 Ribbon 文档，了解属性的具体内容。

注意	上述示例中的 client 应替换为您的 Ribbon 客户端名称。

多个 RestTemplate 对象

如果你想要一个没有负载平衡的 RestTemplate，创建一个 RestTemplate bean 并注入它。要创建@Bean 时，使用@LoadBalanced 限定符来访问负载平衡 RestTemplate。

重要	请注意下面示例中的普通 RestTemplate 声明的@Primary 注释，以消除不合格的

```
@Configuration
public class MyConfiguration {

    @LoadBalanced
    @Bean
    RestTemplate loadBalanced() {
        return new RestTemplate();
    }

    @Primary
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

public class MyClass {
    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    @LoadBalanced
    private RestTemplate loadBalanced;

    public String doOtherStuff() {
        return
loadBalanced.getForObject("http://stores/stores",
String.class);
    }

    public String doStuff() {
        return
restTemplate.getForObject("http://example.com",
String.class);
    }
}
```

提示	如果您看到错误 `java.lang.IllegalArgumentException: Can not set org.springframework.web.client.RestTemplate field com.my.app.F` `com.sun.proxy.$Proxy89`，请尝试注入 `RestOperations` 或设置 `spring.aop`

忽略网络接口

有时，忽略某些命名网络接口是有用的，因此可以将其从服务发现注册中排除（例如，在 Docker 容器中运行）。可以设置正则表达式的列表，这将导致所需的网络接口被忽略。以下配置将忽略“docker0”接口和以“veth”开头的所有接口。

application.yml
```
spring:
  cloud:
    inetutils:
      ignoredInterfaces:
        - docker0
        - veth.*
```

您还可以强制使用正则表达式列表中指定的网络地址：

application.yml
```
spring:
  cloud:
    inetutils:
      preferredNetworks:
        - 192.168
        - 10.0
```

您也可以强制仅使用站点本地地址。有关更多详细信息，请参阅

Inet4Address.html.isSiteLocalAddress（））是什么是站点本地地址。

application.yml
```
spring:
  cloud:
    inetutils:
      useOnlySiteLocalInterfaces: true
```

Spring Cloud Config

Dalston.RELEASE

Spring Cloud Config 为分布式系统中的外部配置提供服务器和客户端支持。使用 Config Server，您可以在所有环境中管理应用程序的外部属性。客户端和服务器上的概念映射与 Spring `Environment` 和 `PropertySource` 抽象相同，因此它们与 Spring 应用程序非常契合，但可以与任何以任何语言运行的应用程序一起使用。随着应用程序通过从开发人员到测试和生产的部署流程，您可以管理这些环境之间的配置，并确定应用程序具有迁移时需要运行的一切。服务器存储后端的默认实现使用 git，因此它轻松支持标签版本的配置环境，以及可以访问用于管理内容的各种工具。很容易添加替代实现，并使用 Spring 配置将其插入。

快速开始

启动服务器：

```
$ cd spring-cloud-config-server
$ ../mvnw spring-boot:run
```

该服务器是一个 Spring Boot 应用程序，所以您可以从 IDE 运行它，而不是喜欢（主类是 `ConfigServerApplication`）。然后尝试一个客户端：

```
$ curl localhost:8888/foo/development
{"name":"development","label":"master","propertySources":
[
  {"name":"https://github.com/scratches/config-repo/foo-
development.properties","source":{"bar":"spam"}},
  {"name":"https://github.com/scratches/config-
repo/foo.properties","source":{"foo":"bar"}}
]}
```

定位资源的默认策略是克隆一个 git 仓库（在 `spring.cloud.config.server.git.uri`），并使用它来初始化一个迷你 `SpringApplication`。小应用程序的 `Environment` 用于枚举属性源并通过 JSON 端点发布。

HTTP 服务具有以下格式的资源：

```
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

其中"应用程序"作为 `SpringApplication` 中的 `spring.config.name` 注入（即常规的 Spring Boot 应用程序中通常是"应用程序"），"配置文件"是活动配置文件（或逗号分隔列表的属性），"label"是可选的 git 标签（默认为"master"）。

Spring Cloud Config 服务器从 git 存储库（必须提供）为远程客户端提供配置：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-
samples/config-repo
```

客户端使用

要在应用程序中使用这些功能，只需将其构建为依赖于 spring-cloud-config-client 的 Spring 引导应用程序（例如，查看配置客户端或示例应用程序的测试用

例）。添加依赖关系的最方便的方法是通过 Spring Boot 启动器 `org.springframework.cloud:spring-cloud-starter-config`。还有一个 Maven 用户的父 pom 和 BOM（`spring-cloud-starter-parent`）和用于 Gradle 和 Spring CLI 用户的 Spring IO 版本管理属性文件。示例 Maven 配置：

的 pom.xml

```
   <parent>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-parent</artifactId>
       <version>1.3.5.RELEASE</version>
       <relativePath /> <!-- lookup parent from repository
-->
   </parent>

<dependencyManagement>
        <dependencies>
                <dependency>

        <groupId>org.springframework.cloud</groupId>
                        <artifactId>spring-cloud-
dependencies</artifactId>
                        <version>Brixton.RELEASE</version>
                        <type>pom</type>
                        <scope>import</scope>
                </dependency>
        </dependencies>
</dependencyManagement>

<dependencies>
        <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-starter-
config</artifactId>
        </dependency>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-
test</artifactId>
                <scope>test</scope>
```

```
        </dependency>
</dependencies>

<build>
        <plugins>
           <plugin>
               <groupId>org.springframework.boot</groupId>
               <artifactId>spring-boot-maven-
plugin</artifactId>
           </plugin>
        </plugins>
</build>

   <!-- repositories also needed for snapshots and
milestones -->
```

那么你可以创建一个标准的 Spring Boot 应用程序，像这个简单的 HTTP 服务器：

```
@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello World!";
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

当它运行它将从端口 8888 上的默认本地配置服务器接收外部配置，如果它正在运行。要修改启动行为，您可以使用`bootstrap.properties`（如`application.properties`）更改配置服务器的位置，但用于应用程序上下文的引导阶段），例如

```
spring.cloud.config.uri: http://myconfigserver.com
```

引导属性将在`/env`端点中显示为高优先级属性源，例如

```
$ curl localhost:8080/env
{
  "profiles":[],
  "configService:https://github.com/spring-cloud-samples/config-repo/bar.properties":{"foo":"bar"},
  "servletContextInitParams":{},
  "systemProperties":{...},
  ...
}
```

（名为"configService：<远程存储库的 URL> / <文件名>"的属性源包含值为"bar"的属性"foo"，是最高优先级）。

注意　属性源名称中的 URL 是 git 存储库，而不是配置服务器 URL。

Spring Cloud Config 服务器

服务器为外部配置（名称值对或等效的 YAML 内容）提供了基于资源的 HTTP。服务器可以使用`@EnableConfigServer`注释轻松嵌入到 Spring Boot 应用程序中。所以这个应用程序是一个配置服务器：

ConfigServer.java

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServer {
  public static void main(String[] args) {
    SpringApplication.run(ConfigServer.class, args);
  }
}
```

像所有的默认端口 8080 上运行的所有 Spring Boot 应用程序一样，但您可以通过各种方式将其切换到常规端口 8888。最简单的也是设置一个默认配置库，它是通过启动它的 `spring.config.name=configserver`（在 Config Server jar 中有一个 `configserver.yml`）。另一个是使用你自己的 `application.properties`，例如

application.properties

```
server.port: 8888
spring.cloud.config.server.git.uri:
file://${user.home}/config-repo
```

其中`${user.home}/config-repo`是包含 YAML 和属性文件的 git 仓库。

注意	在 Windows 中，如果文件 URL 为绝对驱动器前缀，例如 file:///${user.home}/
提示	以下是上面示例中创建 git 仓库的方法： `$ cd $HOME` `$ mkdir config-repo` `$ cd config-repo` `$ git init .` `$ echo info.foo: bar > application.properties` `$ git add -A .` `$ git commit -m "Add application.properties"`
警告	使用本地文件系统进行 git 存储库仅用于测试。使用服务器在生产环境中托管配置
警告	如果您只保留文本文件，则配置库的初始克隆将会快速有效。如果您开始存储二进制 遇到服务器中第一个配置请求和/或内存不足错误的延迟。

环境库

您要在哪里存储配置服务器的配置数据？管理此行为的策略是 `EnvironmentRepository`，服务于 `Environment` 对象。此 `Environment` 是

Spring `Environment`（包括 `propertySources` 作为主要功能）的域的浅层副本。`Environment` 资源由三个变量参数化：

- `{application}`映射到客户端的"spring.application.name"；
- `{profile}`映射到客户端上的"spring.profiles.active"（逗号分隔列表）；和
- `{label}`这是一个服务器端功能，标记"版本"的配置文件集。

存储库实现通常表现得像一个 Spring Boot 应用程序从"spring.config.name"等于`{application}`参数加载配置文件，"spring.profiles.active"等于`{profiles}`参数。配置文件的优先级规则也与常规启动应用程序相同：活动配置文件优先于默认配置，如果有多个配置文件，则最后一个获胜（例如向 `Map` 添加条目）。

示例：客户端应用程序具有此引导配置：

bootstrap.yml

```
spring:
  application:
    name: foo
  profiles:
    active: dev,mysql
```

（通常使用 Spring Boot 应用程序，这些属性也可以设置为环境变量或命令行参数）。

如果存储库是基于文件的，则服务器将从 `application.yml` 创建 `Environment`（在所有客户端之间共享），`foo.yml`（以 `foo.yml` 优先））。如果 YAML 文件中有文件指向 Spring 配置文件，那么应用的优先级更高（按照列出的配置文件的顺序），并且如果存在特定于配置文件的 YAML（或属性）文

件，那么这些文件也应用于优先级高于默认值。较高优先级转换为 `Environment` 之前列出的 `PropertySource`。（这些规则与独立的 Spring Boot 应用程序相同。）

Git 后端

`EnvironmentRepository` 的默认实现使用 Git 后端，这对于管理升级和物理环境以及审核更改非常方便。要更改存储库的位置，可以在 Config Server 中设置“spring.cloud.config.server.git.uri”配置属性（例如 `application.yml`）。如果您使用 `file:` 前缀进行设置，则应从本地存储库中工作，以便在没有服务器的情况下快速方便地启动，但在这种情况下，服务器将直接在本地存储库上进行操作，而不会克隆如果它不是裸机，因为配置服务器永远不会更改“远程”资源库）。要扩展 Config Server 并使其高度可用，您需要将服务器的所有实例指向同一个存储库，因此只有共享文件系统才能正常工作。即使在这种情况下，最好使用共享文件系统存储库的 `ssh:` 协议，以便服务器可以将其克隆并使用本地工作副本作为缓存。

该存储库实现将 HTTP 资源的 `{label}` 参数映射到 git 标签（提交 ID，分支名称或标签）。如果 git 分支或标签名称包含斜杠（“/”），则应使用特殊字符串“（_）”指定 HTTP URL 中的标签，以避免与其他 URL 路径模糊。例如，如果标签为 `foo/bar`，则替换斜杠将导致标签看起来像 `foo(_)bar`。如果您使用像 curl 这样的命令行客户端（例如使用引号将其从 shell 中转出来），请小心 URL 中的方括号。

Git URI 中的占位符

Spring Cloud Config 服务器支持一个 Git 仓库 URL，其中包含`{application}`和`{profile}`（以及`{label}`）的占位符，如果需要，请记住标签应用为 git 标签）。因此，您可以使用（例如）轻松支持“每个应用程序的一个 repo”策略：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/myorg/{application}
```

或使用类似模式但使用`{profile}`的“每个配置文件一个”策略。

模式匹配和多个存储库

还可以通过应用程序和配置文件名称的模式匹配来支持更复杂的需求。模式格式是带有通配符的`{application}/{profile}`名称的逗号分隔列表（可能需要引用以通配符开头的模式）。例：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-
samples/config-repo
          repos:
            simple: https://github.com/simple/config-repo
            special:
              pattern: special*/dev*,*special*/dev*
              uri: https://github.com/special/config-repo
            local:
              pattern: local*
              uri: file:/home/configsvc/config-repo
```

如果`{application}/{profile}`不匹配任何模式，它将使用在“spring.cloud.config.server.git.uri”下定义的默认 uri。在上面的例子中，对于“简单”存储库，模式是`simple/*`（即所有配置文件中只匹配一个名为“简单”的应用程序）。“本地”存储库与所有配置文件中以“local”开头的所有应用程序名称匹配（将`/*`后缀自动添加到任何没有配置文件匹配器的模式）。

注意	在上述“简单”示例中使用的“单行”快捷方式只能在唯一要设置的属性为 URI 的情 内容（凭据，模式等），则需要使用完整的表单。

repo 中的`pattern`属性实际上是一个数组，因此您可以使用属性文件中的 YAML 数组（或`[0]`，`[1]`等后缀）绑定到多个模式。如果要运行具有多个配置文件的应用程序，则可能需要执行此操作。例：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-
samples/config-repo
          repos:
            development:
              pattern:
                - */development
                - */staging
              uri: https://github.com/development/config-
repo
            staging:
              pattern:
                - */qa
                - */production
              uri: https://github.com/staging/config-repo
```

注意 Spring Cloud 将猜测包含不在*中的配置文件的模式意味着您实际上要匹配从此模式
/staging 是["/staging", "*/staging,*"]）。这是常见的，您需要在本地
序，但也可以远程运行“云”配置文件。

每个存储库还可以选择将配置文件存储在子目录中，搜索这些目录的模式可以指定为 `searchPaths`。例如在顶层：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-
samples/config-repo
          searchPaths: foo,bar*
```

在此示例中，服务器搜索顶级和“foo /”子目录以及名称以“bar”开头的任何子目录中的配置文件。

默认情况下，首次请求配置时，服务器克隆远程存储库。服务器可以配置为在启动时克隆存储库。例如在顶层：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://git/common/config-repo.git
          repos:
            team-a:
                pattern: team-a-*
                cloneOnStart: true
                uri: http://git/team-a/config-repo.git
            team-b:
                pattern: team-b-*
                cloneOnStart: false
                uri: http://git/team-b/config-repo.git
            team-c:
```

```
            pattern: team-c-*
            uri: http://git/team-a/config-repo.git
```

在此示例中，服务器在启动之前克隆了 team-a 的 config-repo，然后它接受任何请求。所有其他存储库将不被克隆，直到请求从存储库配置。

注意	在配置服务器启动时设置要克隆的存储库可以帮助在配置服务器启动时快速识别错误 URI）。配置源不启用 `cloneOnStart` 时，配置服务器可能会成功启动配置错误或无 到应用程序从该配置源请求配置为止。

认证

要在远程存储库上使用 HTTP 基本身份验证，请分别添加“username”和“password”属性（不在 URL 中），例如

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-
samples/config-repo
          username: trolley
          password: strongpassword
```

如果您不使用 HTTPS 和用户凭据，当您将密钥存储在默认目录（`~/.ssh`）中，并且 uri 指向 SSH 位置时，SSH 也应该开箱即用，例如“ git@github.com：配置/云配置”。必须在`~/.ssh/known_hosts`文件中存在 Git 服务器的条目，并且它是 `ssh-rsa` 格式。其他格式（如 `ecdsa-sha2-nistp256`）不受支持。为了避免意外，您应该确保 Git 服务器的 `known_hosts` 文件中只有一个条目，并且与您提供给配置服务器的 URL 匹配。如果您在 URL 中使用了主机名，那么您希望在 `known_hosts` 文件中具有这一点，而不是 IP。使用 JGit 访问存储库，因此您

发现的任何文档都应适用。HTTPS 代理设置可以`~/.git/config`设置，也可以通过系统属性（`-Dhttps.proxyHost`和`-Dhttps.proxyPort`）与任何其他 JVM 进程相同。

提示	如果您不知道`~/.git`目录使用`git config --global`来处理设置的位置（例如`http.sslVerify false`）。

使用 AWS CodeCommit 进行认证

AWS CodeCommit 认证也可以完成。当从命令行使用 Git 时，AWS CodeCommit 使用身份验证助手。该帮助器不与 JGit 库一起使用，因此如果 Git URI 与 AWS CodeCommit 模式匹配，则将创建用于 AWS CodeCommit 的 JGit CredentialProvider。AWS CodeCommit URI 始终看起来像 https：//git-codecommit.$ {AWS_REGION} .amazonaws.com / $ {repopath}。

如果您使用 AWS CodeCommit URI 提供用户名和密码，那么这些 URI 必须 是用于访问存储库的 AWS accessKeyId 和 secretAccessKey。如果不指定用户名和密码，则将使用 AWS 默认凭据提供程序链检索 accessKeyId 和 secretAccessKey 。

如果您的 Git URI 与 CodeCommit URI 模式（上述）匹配，则必须在用户名和密码或默认凭据提供程序链支持的某个位置中提供有效的 AWS 凭据。AWS EC2 实例可以使用 EC2 实例的 IAM 角色。

注意：aws-java-sdk-core jar 是一个可选的依赖关系。如果 aws-java-sdk-core jar 不在您的类路径上，则无论 git 服务器 URI 如何，都将不会创建 AWS 代码提交凭据提供程序。

Git 搜索路径中的占位符

Spring Cloud Config 服务器还支持具有`{application}`和`{profile}`（以及`{label}`（如果需要））占位符的搜索路径。例:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-
samples/config-repo
          searchPaths: '{application}'
```

在资源库中搜索与目录（以及顶级）相同名称的文件。通配符在具有占位符的搜索路径中也是有效的（搜索中包含任何匹配的目录）。

力拉入 Git 存储库

如前所述 Spring Cloud Config 服务器克隆远程 git 存储库，如果某种方式本地副本变脏（例如，通过操作系统进程更改文件夹内容），则 Spring Cloud Config 服务器无法从远程存储库更新本地副本。

要解决这个问题，有一个`force-pull`属性，如果本地副本是脏的，将使 Spring Cloud Config Server 强制从远程存储库拉。例:

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-
samples/config-repo
          force-pull: true
```

如果您有多个存储库配置，则可以为每个存储库配置 `force-pull` 属性。例：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://git/common/config-repo.git
          force-pull: true
          repos:
            team-a:
                pattern: team-a-*
                uri: http://git/team-a/config-repo.git
                force-pull: true
            team-b:
                pattern: team-b-*
                uri: http://git/team-b/config-repo.git
                force-pull: true
            team-c:
                pattern: team-c-*
                uri: http://git/team-a/config-repo.git
```

注意	`force-pull` 属性的默认值为 `false`。

版本控制后端文件系统使用

警告	使用基于 VCS 的后端（git，svn）文件被检出或克隆到本地文件系统。默认情况下， `config-repo-`。在 linux 上，例如可以是 `/tmp/config-repo-<randomid>`。一 可能会导致意外的行为，例如缺少属性。为避免此问题，请通过将 `spring.cloud` `spring.cloud.config.server.svn.basedir` 设置为不驻留在系统临时结构中 录。

文件系统后端

配置服务器中还有一个不使用 Git 的“本机”配置文件，只是从本地类路径或文件系统加载配置文件（您想要指向的任何静态 URL“spring.cloud.config.server .native.searchLocations“）。要使用本机配置文件，只需使用“spring.profiles.active = native”启动 Config Server。

注意	请记住使用 `file:`前缀的文件资源（缺省没有前缀通常是 classpath）。与任何 Sp 式的环境占位符，但请记住，Windows 中的绝对路径需要额外的“/”，例如 file:
警告	`searchLocations` 的默认值与本地 Spring Boot 应用程序（所以`[classpath:/, file:./config]`）相同。这不会将 `application.properties` 从服务器暴露给 前，服务器中存在的任何属性源都将被删除。
提示	文件系统后端对于快速入门和测试是非常好的。要在生产中使用它，您需要确保文 有实例中共享。

搜索位置可以包含`{application}`，`{profile}`和`{label}`的占位符。以这种方式，您可以隔离路径中的目录，并选择一个有用的策略（例如每个应用程序的子目录或每个配置文件的子目录）。

如果您不在搜索位置使用占位符，则该存储库还将 HTTP 资源的`{label}`参数附加到搜索路径上的后缀，因此属性文件将从每个搜索位置加载**并**具有相同名称的子目录作为标签（标记的属性在 Spring 环境中优先）。因此，没有占位符的默认行为与添加以`/{label}/. For example `file:/tmp/config`结尾的搜索位置与 `file:/tmp/config,file:/tmp/config/{label}`相同

Vault 后端

Spring Cloud Config 服务器还支持 Vault 作为后端。

Vault 是安全访问秘密的工具。一个秘密是你想要严格控制访问的任何东西，如 API 密钥，密码，证书等等。Vault 为任何秘密提供统一的界面，同时提供严格的访问控制和记录详细的审核日志。

有关 Vault 的更多信息，请参阅 Vault 快速入门指南。

要使配置服务器使用 Vault 后端，您必须使用 `vault` 配置文件运行配置服务器。例如在配置服务器的 `application.properties` 中，您可以添加 `spring.profiles.active=vault`。

默认情况下，配置服务器将假定您的 Vault 服务器正在运行于 http://127.0.0.1:8200。它还将假定后端名称为 `secret`，密钥为 `application`。所有这些默认值都可以在配置服务器的 `application.properties` 中配置。以下是可配置 Vault 属性的表。所有属性前缀为 `spring.cloud.config.server.vault`。

名称	默认值
host	127.0.0.1
port	8200
scheme	HTTP
backend	秘密
defaultKey	应用
profileSeparator	,

所有可配置的属性可以在 `org.springframework.cloud.config.server.environment.VaultEnvironmentRepository` 找到。

运行配置服务器后，可以向服务器发出 HTTP 请求，以从 Vault 后端检索值。为此，您需要为 Vault 服务器创建一个令牌。

首先放置一些数据给你 Vault。例如

```
$ vault write secret/application foo=bar baz=bam
$ vault write secret/myapp foo=myappsbar
```

现在，将 HTTP 请求发送给您的配置服务器以检索值。

```
$ curl -X "GET" "http://localhost:8888/myapp/default" -H "X-Config-Token: yourtoken"
```

在提出上述要求后，您应该会看到类似的回复。

```
{
   "name":"myapp",
   "profiles":[
      "default"
   ],
   "label":null,
   "version":null,
   "state":null,
   "propertySources":[
      {
         "name":"vault:myapp",
         "source":{
            "foo":"myappsbar"
         }
      },
      {
         "name":"vault:application",
         "source":{
            "baz":"bam",
            "foo":"bar"
         }
      }
   ]
}
```

多个 Properties 来源

使用 Vault 时，您可以为应用程序提供多个属性源。例如，假设您已将数据写入 Vault 中的以下路径。

```
secret/myApp,dev
secret/myApp
secret/application,dev
secret/application
```

写入 `secret/application` 的 Properties 可 用于使用配置服务器的所有应用程序。名称为 `myApp` 的应用程序将具有写入 `secret/myApp` 和 `secret/application` 的任何属性。当 `myApp` 启用 `dev` 配置文件时，写入所有上述路径的属性将可用，列表中第一个路径中的属性优先于其他路径。

与所有应用共享配置

基于文件的存储库

使用基于文件（即 git，svn 和 native）的存储库，文件名为 `application*` 的资源在所有客户端应用程序（所以 `application.properties`，`application.yml`，`application-*.properties` 等）之间共享）。您可以使用这些文件名的资源来配置全局默认值，并根据需要将其覆盖应用程序特定的文件。

#_property_overrides [属性覆盖]功能也可用于设置全局默认值，并且允许占位符应用程序在本地覆盖它们。

提示	使用“本机”配置文件（本地文件系统后端），建议您使用不属于服务器自身配置的 中的 `application*` 资源将被删除，因为它们是服务器的一部分。

Vault 服务器

当使用 Vault 作为后端时，可以通过将配置放在 `secret/application` 中与所有应用程序共享配置。例如，如果您运行此 Vault 命令

```
$ vault write secret/application foo=bar baz=bam
```

使用配置服务器的所有应用程序都可以使用属性 `foo` 和 `baz`。

复合环境库

在某些情况下，您可能希望从多个环境存储库中提取配置数据。为此，只需在配置服务器的应用程序属性或 YAML 文件中启用多个配置文件即可。例如，如果您要从 Git 存储库以及 SVN 存储库中提取配置数据，那么您将为配置服务器设置以下属性。

```
spring:
  profiles:
    active: git, svn
  cloud:
    config:
      server:
        svn:
          uri: file:///path/to/svn/repo
          order: 2
        git:
          uri: file:///path/to/git/repo
          order: 1
```

除了指定 URI 的每个 repo 之外，还可以指定 `order` 属性。`order` 属性允许您指定所有存储库的优先级顺序。`order` 属性的数值越低，优先级越高。存储库的优先顺序将有助于解决包含相同属性的值的存储库之间的任何潜在冲突。

注意	从环境仓库检索值时的任何类型的故障将导致整个复合环境的故障。
注意	当使用复合环境时，重要的是所有 repos 都包含相同的标签。如果您有类似于上述的配置数据，但是 SVN repo 不包含称为 `master` 的分支，则整个请求将失败。

自定义复合环境库

除了使用来自 Spring Cloud 的环境存储库之外，还可以提供自己的 `EnvironmentRepository` bean 作为复合环境的一部分。要做到这一点，你的 bean 必须实现 `EnvironmentRepository` 接口。如果要在复合环境中控制自定义 `EnvironmentRepository` 的优先级，您还应该实现 `Ordered` 接口并覆盖 `getOrdered` 方法。如果您不实现 `Ordered` 接口，那么您的 `EnvironmentRepository` 将被赋予最低优先级。

属性覆盖

配置服务器具有"覆盖"功能，允许操作员为应用程序使用普通的 Spring Boot 钩子不会意外更改的所有应用程序提供配置属性。要声明覆盖，只需将名称/值对的地图添加到 `spring.cloud.config.server.overrides`。例如

```
spring:
  cloud:
    config:
      server:
        overrides:
          foo: bar
```

将导致配置客户端的所有应用程序独立于自己的配置读取 `foo=bar`。（当然，应用程序可以以任何方式使用 Config Server 中的数据，因此覆盖不可强制执

行，但如果它们是 Spring Cloud Config 客户端，则它们确实提供有用的默认行为。）

提示	通过使用反斜杠（“\”）来转义“$”或“{”，例如\${app.foo:bar}解析，可环境占位符到“bar”，除非应用程序提供自己的“app.foo”。请注意，在 YAML 中执行的属性文件中配置服务器上的覆盖。

您可以通过在远程存储库中设置标志 `spring.cloud.config.overrideNone=true`（默认为 false），将客户端中所有覆盖的优先级更改为更为默认值，允许应用程序在环境变量或系统属性中提供自己的值。

健康指标

配置服务器附带运行状况指示器，检查配置的 `EnvironmentRepository` 是否正常工作。默认情况下，它要求 `EnvironmentRepository` 应用程序名称为 `app`，`default` 配置文件和 `EnvironmentRepository` 实现提供的默认标签。

您可以配置运行状况指示器以检查更多应用程序以及自定义配置文件和自定义标签，例如

```
spring:
  cloud:
    config:
      server:
        health:
          repositories:
            myservice:
              label: mylabel
            myservice-dev:
              name: myservice
```

```
            profiles: development
```

您可以通过设置 `spring.cloud.config.server.health.enabled=false` 来禁用运行状况指示器。

安全

您可以以任何对您有意义的方式（从物理网络安全性到 OAuth2 承载令牌）保护您的 Config Server，并且 Spring Security 和 Spring Boot 可以轻松做任何事情。

要使用默认的 Spring Boot 配置的 HTTP Basic 安全性，只需在类路径中包含 Spring Security（例如通过 `spring-boot-starter-security`）。默认值为“user”的用户名和随机生成的密码，这在实践中不会非常有用，因此建议您配置密码（通过 `security.user.password`）并对其进行加密（请参阅下文的说明怎么做）。

加密和解密

重要	**先决条件：** 要使用加密和解密功能，您需要在 JVM 中安装全面的 JCE（默认情况下不 加密扩展（JCE）无限强度管理策略文件”，并按照安装说明（实际上将 JRE lib / 为您下载的文件）。

如果远程属性源包含加密内容（以`{cipher}`开头的值），则在通过 HTTP 发送到客户端之前，它们将被解密。这种设置的主要优点是，当它们“静止”时，属性值不必是纯文本（例如在 git 仓库中）。如果值无法解密，则从属性源中删除该值，并添加具有相同键的附加属性，但以“无效”作为前缀。和“不适用”的值（通常为“<n / a>”）。这主要是为了防止密码被用作密码并意外泄漏。

如果要为 config 客户端应用程序设置远程配置存储库，可能会包含一个 `application.yml`，例如：

application.yml

```
spring:
  datasource:
    username: dbuser
    password: '{cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ'
```

.properties 文件中的加密值不能用引号括起来，否则不会解密该值：

application.properties

```
spring.datasource.username: dbuser
spring.datasource.password:
{cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ
```

您可以安全地将此纯文本推送到共享 git 存储库，并且保密密码。

服务器还暴露了`/encrypt`和`/decrypt`端点（假设这些端点将被保护，并且只能由授权代理访问）。如果您正在编辑远程配置文件，可以使用 Config Server 通过 POST 到`/encrypt`端点来加密值，例如

```
$ curl localhost:8888/encrypt -d mysecret
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a
49f7bda
```

注意	如果要加密的值具有需要进行 URL 编码的字符，则应使用`--data-urlencode`选项

逆向操作也可通过`/decrypt`获得（如果服务器配置了对称密钥或全密钥对）：

```
$ curl localhost:8888/decrypt -d
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a
49f7bda
mysecret
```

提示	如果您使用 `curl` 进行测试，则使用`--data-urlencode`（而不是`-d`）或设置显式 保在有特殊字符时正确地对数据进行编码（'+'特别是棘手）。

将加密的值添加到`{cipher}`前缀，然后再将其放入 YAML 或属性文件中，然后再提交并将其推送到远程可能不安全的存储区。

`/encrypt`和`/decrypt`端点也都接受`/*/{name}/{profiles}`形式的路径，当客户端调用到主环境资源时，可以用于每个应用程序（名称）和配置文件控制密码。

注意	为了以这种细微的方式控制密码，您还必须提供一种`TextEncryptorLocator`类件创建不同的加密器。默认提供的不会这样做（所有加密使用相同的密钥）。

`spring`命令行客户端（安装了 Spring Cloud CLI 扩展）也可以用于加密和解密，例如

```
$ spring encrypt mysecret --key foo
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a
49f7bda
$ spring decrypt --key foo
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a
49f7bda
mysecret
```

要在文件中使用密钥（例如用于加密的 RSA 公钥），使用"@"键入键值，并提供文件路径，例如

```
$ spring encrypt mysecret --key @${HOME}/.ssh/id_rsa.pub
AQAjPgt3eFZQXwt8tsHAVv/QHiY5sI2dRcR+...
```

关键参数是强制性的（尽管有一个`--`前缀）。

密钥管理

Config Server 可以使用对称（共享）密钥或非对称密钥（RSA 密钥对）。非对称选择在安全性方面是优越的，但是使用对称密钥往往更方便，因为它只是配置的一个属性值。

要配置对称密钥，您只需要将 `encrypt.key` 设置为一个秘密字符串（或使用环境变量 `ENCRYPT_KEY` 将其从纯文本配置文件中删除）。

要配置非对称密钥，您可以将密钥设置为 PEM 编码的文本值（`encrypt.key`），也可以通过密钥库设置密钥（例如由 JDK 附带的 `keytool` 实用程序创建）。密钥库属性为 `encrypt.keyStore.*`，`*`等于

- `location`（a `Resource` 位置），
- `password`（解锁密钥库）和
- `alias`（以识别商店中使用的密钥）。

使用公钥进行加密，需要私钥进行解密。因此，原则上您只能在服务器中配置公钥，如果您只想进行加密（并准备使用私钥本地解密值）。实际上，您可能不想这样做，因为它围绕所有客户端传播密钥管理流程，而不是将其集中在服务器中。另一方面，如果您的配置服务器真的相对不安全，并且只有少数客户端需要加密的属性，这是一个有用的选项。

创建用于测试的密钥库

要创建一个密钥库进行测试，您可以执行以下操作：

```
$ keytool -genkeypair -alias mytestkey -keyalg RSA \
  -dname "CN=Web
Server,OU=Unit,O=Organization,L=City,S=State,C=US" \
  -keypass changeme -keystore server.jks -storepass
letmein
```

将 `server.jks` 文件放在类路径（例如）中，然后在您的 `application.yml` 中配置服务器：

```
encrypt:
  keyStore:
    location: classpath:/server.jks
    password: letmein
    alias: mytestkey
    secret: changeme
```

使用多个键和键旋转

除了加密属性值中的`{cipher}`前缀之外，配置服务器在（Base64 编码）密文开始前查找`{name:value}`前缀（零或多个）。密钥被传递给 `TextEncryptorLocator`，它可以执行找到密码的 `TextEncryptor` 所需的任何逻辑。如果配置了密钥库（`encrypt.keystore.location`），默认定位器将使用"key"前缀提供的别名，即使用如下密码查找存储中的密钥：

```
foo:
  bar: `{cipher}{key:testkey}...`
```

定位器将寻找一个名为"testkey"的键。也可以通过前缀中的`{secret:…}`值提供一个秘密，但是如果不是默认值，则使用密钥库密码（这是您在构建密钥库时获得的，并且不指定密码）。如果你**这样做** 提供一个秘密建议你也加密使用自定义 `SecretLocator` 的秘密。

如果密钥只用于加密几个字节的配置数据（即它们没有在其他地方使用），则密码转换几乎不是必需的，但是如果存在安全漏洞，有时您可能需要更改密钥实例。在这种情况下，所有客户端都需要更改其源配置文件（例如，以 git 格式），并在所有密码中使用新的`{key:…}`前缀，当然事先检查密钥别名在配置服务器密钥库中是否可用。

提示	如果要让 Config Server 处理所有加密以及解密，也可以将`{name:value}`前缀添

服务加密 Properties

有时您希望客户端在本地解密配置，而不是在服务器中进行配置。在这种情况下，您仍然可以拥有/加密和解密端点（如果您提供`encrypt.*`配置来定位密钥），但是您需要使用`spring.cloud.config.server.encrypt.enabled=false`明确地关闭传出属性的解密。如果您不关心端点，那么如果您既不配置密钥也不配置使能的标志，则应该起作用。

服务替代格式

来自环境端点的默认 JSON 格式对于 Spring 应用程序的消费是完美的，因为它直接映射到`Environment`抽象。如果您喜欢，可以通过向资源路径（“.yml”，“.yaml”或“.properties”）添加后缀来使用与 YAML 或 Java 属性相同的数据。这对于不关心 JSON 端点的结构的应用程序或其提供的额外的元数据的应用程序来说

可能是有用的，例如，不使用 Spring 的应用程序可能会受益于此方法的简单性。

YAML 和属性表示有一个额外的标志（作为一个布尔查询参数 `resolvePlaceholders` 提供））,以标示 Spring `${…}`形式的源文档中的占位符，应在输出中解析可能在渲染之前。对于不了解 Spring 占位符惯例的消费者来说，这是一个有用的功能。

注意	使用 YAML 或属性格式存在局限性，主要是与元数据的丢失有关。JSON 被构造为属性联。即使源的起源具有多个源，并且原始源文件的名称丢失，YAML 和属性表也合并储库中 YAML 源的忠实表示：它是由平面属性源的列表构建的，并且必须对键的形式

服务纯文本

您的应用程序可能需要通用的纯文本配置文件，而不是使用 `Environment` 抽象（或 YAML 中的其他替代表示形式或属性格式）。配置服务器通过 `/{name}/{profile}/{label}/{path}`附加的端点提供这些服务，其中“name”，“profile”和“label”的含义与常规环境端点相同，但“path”是文件名（例如 `log.xml`）。此端点的源文件位于与环境端点相同的方式：与属性或 YAML 文件相同的搜索路径，而不是聚合所有匹配的资源，只返回匹配的第一个。

找到资源后，使用正确格式（`${…}`）的占位符将使用有效的 `Environment` 解析为应用程序名称，配置文件和标签提供。以这种方式，资源端点与环境端点紧密集成。例如，如果您有一个 GIT（或 SVN）资源库的布局：

```
application.yml
```

```
nginx.conf
```

其中`nginx.conf`如下所示:

```
server {
    listen              80;
    server_name         ${nginx.server.name};
}
```

和`application.yml`这样:

```
nginx:
  server:
    name: example.com
---
spring:
  profiles: development
nginx:
  server:
    name: develop.com
```

那么`/foo/default/master/nginx.conf`资源如下所示:

```
server {
    listen              80;
    server_name         example.com;
}
```

和`/foo/development/master/nginx.conf`这样:

```
server {
    listen              80;
    server_name         develop.com;
}
```

注意	就像环境配置的源文件一样，“配置文件”用于解析文件名，因此，如果您想要一个 `/*/development/*/logback.xml`将由一个名为`logback-development.xml` `logback.xml`）。

嵌入配置服务器

配置服务器最好作为独立应用程序运行，但如果需要，可以将其嵌入到另一个应用程序中。只需使用`@EnableConfigServer`注释。在这种情况下可以使用的可选属性是`spring.cloud.config.server.bootstrap`，它是一个标志，表示服务器应该从其自己的远程存储库配置自身。该标志默认关闭，因为它可能会延迟启动，但是当嵌入在另一个应用程序中时，以与其他应用程序相同的方式初始化是有意义的。

注意	应该是显而易见的，但请记住，如果您使用引导标志，配置服务器将需要在`boots` URI。

要更改服务器端点的位置，您可以（可选）设置`spring.cloud.config.server.prefix`，例如“/ config”，以提供前缀下的资源。前缀应该开始但不以“/”结尾。它被应用于配置服务器中的`@RequestMappings`（即 Spring Boot 前缀`server.servletPath`和`server.contextPath`）之下。

如果您想直接从后端存储库（而不是从配置服务器）读取应用程序的配置，这基本上是一个没有端点的嵌入式配置服务器。如果不使用`@EnableConfigServer`注释（仅设置`spring.cloud.config.server.bootstrap=true`），则可以完全关闭端点。

推送通知和 Spring Cloud Bus

许多源代码存储库提供程序（例如 Github，Gitlab 或 Bitbucket）将通过 webhook 通知您存储库中的更改。您可以通过提供商的用户界面将 webhook 配置为 URL 和一组感兴趣的事件。例如，Github 将使用包含提交列表的 JSON 主体和“X-Github-Event”等于“push”的头文件发送到 webhook。如果在 `spring-cloud-config-monitor` 库中添加依赖关系并激活配置服务器中的 Spring Cloud Bus，则启用“/ monitor”端点。

当 Webhook 被激活时，配置服务器将发送一个 `RefreshRemoteApplicationEvent` 针对他认为可能已经改变的应用程序。变更检测可以进行策略化，但默认情况下，它只是查找与应用程序名称匹配的文件的更改（例如，“foo.properties”针对的是“foo”应用程序，“application.properties”针对所有应用程序）。如果要覆盖该行为的策略是 `PropertyPathNotificationExtractor`，它接受请求标头和正文作为参数，并返回更改的文件路径列表。

默认配置与 Github，Gitlab 或 Bitbucket 配合使用。除了来自 Github，Gitlab 或 Bitbucket 的 JSON 通知之外，您还可以通过使用表单编码的身体参数 `path={name}` 通过 POST 为“/ monitor”来触发更改通知。这将广播到匹配“{name}”模式的应用程序（可以包含通配符）。

注意	只有在配置服务器和客户端应用程序中激活 `spring-cloud-bus` 时才会传送 `Refr`

注意	默认配置还检测本地 git 存储库中的文件系统更改（在这种情况下不使用 webhook，新）。

Spring Cloud Config 客户端

Spring Boot 应用程序可以立即利用 Spring 配置服务器（或应用程序开发人员提供的其他外部属性源），并且还将获取与 `Environment` 更改事件相关的一些其他有用功能。

配置第一引导

这是在类路径上具有 Spring Cloud Config Client 的任何应用程序的默认行为。配置客户端启动时，它将通过配置服务器（通过引导配置属性 `spring.cloud.config.uri`）绑定，并使用远程属性源初始化 Spring `Environment`。

这样做的最终结果是所有想要使用 Config Server 的客户端应用程序需要 `bootstrap.yml`（或环境变量），服务器地址位于 `spring.cloud.config.uri`（默认为“http：// localhost：8888”）。

发现第一个引导

如果您正在使用 DiscoveryClient 实现，例如 Spring Cloud Netflix 和 Eureka 服务发现或 Spring Cloud Consul（Spring Cloud Zookeeper 不支持此功能），那么您

可以使用 Config Server 如果您想要发现服务注册，但在默认的“配置优先”模式下，客户端将无法利用注册。

如果您希望使用 `DiscoveryClient` 找到配置服务器，可以通过设置 `spring.cloud.config.discovery.enabled=true`（默认为“false”）来实现。最终的结果是，客户端应用程序都需要具有适当发现配置的 `bootstrap.yml`（或环境变量）。例如，使用 Spring Cloud Netflix，您需要定义 Eureka 服务器地址，例如 `eureka.client.serviceUrl.defaultZone`。使用此选项的价格是启动时额外的网络往返，以定位服务注册。好处是配置服务器可以更改其坐标，只要发现服务是一个固定点。默认的服务标识是“configserver”，但您可以使用 `spring.cloud.config.discovery.serviceId` 在客户端进行更改（在服务器上以服务的通常方式更改，例如设置 `spring.application.name`）。

发现客户端实现都支持某种元数据映射（例如 Eureka，我们有 `eureka.instance.metadataMap`）。可能需要在其服务注册元数据中配置 Config Server 的一些其他属性，以便客户端可以正确连接。如果使用 HTTP Basic 安全配置服务器，则可以将凭据配置为“用户名”和“密码”。并且如果配置服务器具有上下文路径，您可以设置“configPath”。例如，对于作为 Eureka 客户端的配置服务器：

bootstrap.yml

```
eureka:
  instance:
    ...
    metadataMap:
      user: osufhalskjrtl
```

```
    password: lviuhlszvaorhvlo5847
    configPath: /config
```

配置客户端快速失败

在某些情况下，如果服务无法连接到配置服务器，则可能希望启动服务失败。如果这是所需的行为，请设置引导配置属性 `spring.cloud.config.failFast=true`，客户端将以异常停止。

配置客户端重试

如果您希望配置服务器在您的应用程序启动时可能偶尔不可用，您可以要求它在发生故障后继续尝试。首先，您需要设置 `spring.cloud.config.failFast=true`，然后您需要添加 `spring-retry` 和 `spring-boot-starter-aop` 到您的类路径。默认行为是重试 6 次，初始退避间隔为 1000ms，指数乘数为 1.1，用于后续退避。您可以使用 `spring.cloud.config.retry.*`配置属性配置这些属性（和其他）。

提示	要完全控制重试，请使用 ID “configServerRetryInterceptor” 添加 `RetryOpera` Spring 重试有一个 `RetryInterceptorBuilder` 可以轻松创建一个。

查找远程配置资源

配置服务从`/{name}/{profile}/{label}`提供属性源，客户端应用程序中的默认绑定

- “name”= `${spring.application.name}`

- “profile”= `${spring.profiles.active}` （实际上是 `Environment.getActiveProfiles()`）
- “label”=“master”

所有这些都可以通过设置 `spring.cloud.config.*` （其中`*`是“name”, “profile”或“label”）来覆盖。“标签”可用于回滚到以前版本的配置; 使用默认的 Config Server 实现，它可以是 git 标签，分支名称或提交 ID。标签也可以以逗号分隔的列表形式提供，在这种情况下，列表中的项目会逐个尝试，直到成功。例如，当您可能希望将配置标签与您的分支对齐，但使其成为可选（例如 `spring.cloud.config.label=myfeature,develop`）时，这对于在特征分支上工作时可能很有用。

安全

如果您在服务器上使用 HTTP 基本安全性，那么客户端只需要知道密码（如果不是默认用户名）。您可以通过配置服务器 URI，或通过单独的用户名和密码属性，例如

bootstrap.yml
```
spring:
  cloud:
    config:
     uri: https://user:secret@myconfig.mycompany.com
```

要么

bootstrap.yml
```
spring:
  cloud:
```

```
    config:
     uri: https://myconfig.mycompany.com
     username: user
     password: secret
```

`spring.cloud.config.password`和`spring.cloud.config.username`值覆盖 URI 中提供的任何内容。

如果您在 Cloud Foundry 部署应用程序，则提供密码的最佳方式是通过服务凭证（例如 URI），因为它甚至不需要在配置文件中。在 Cloud Foundry 上为本地工作的用户提供的服务的一个例子，名为"configserver"：

bootstrap.yml

```
spring:
  cloud:
    config:
     uri:
${vcap.services.configserver.credentials.uri:http://user:
password@localhost:8888}
```

如果您使用另一种形式的安全性，则可能需要向`ConfigServicePropertySourceLocator`提供`RestTemplate`（例如，通过在引导上下文中获取它并注入一个）。`ConfigServicePropertySourceLocator`提供{470 /}（例如通过在引导上下文中获取它并注入）。

健康指标

Config Client 提供一个尝试从 Config Server 加载配置的 Spring Boot 运行状况指示器。可以通过设置`health.config.enabled=false`来禁用运行状况指示

器。由于性能原因，响应也被缓存。默认缓存生存时间为 5 分钟。要更改该值，请设置`health.config.time-to-live`属性（以毫秒为单位）。

提供自定义 RestTemplate

在某些情况下，您可能需要从客户端自定义对配置服务器的请求。通常这涉及传递特殊的`Authorization`标头来对服务器的请求进行身份验证。要提供自定义`RestTemplate`，请按照以下步骤操作。

1. 设置`spring.cloud.config.enabled=false`以禁用现有的配置服务器属性源。

2. 使用`PropertySourceLocator`实现创建一个新的配置 bean。

CustomConfigServiceBootstrapConfiguration.java

```
@Configuration
public class CustomConfigServiceBootstrapConfiguration {
    @Bean
    public ConfigClientProperties configClientProperties()
{
        ConfigClientProperties client = new
ConfigClientProperties(this.environment);
        client.setEnabled(false);
        return client;
    }

    @Bean
    public ConfigServicePropertySourceLocator
configServicePropertySourceLocator() {
        ConfigClientProperties clientProperties =
configClientProperties();
       ConfigServicePropertySourceLocator
configServicePropertySourceLocator =  new
ConfigServicePropertySourceLocator(clientProperties);

configServicePropertySourceLocator.setRestTemplate(custom
RestTemplate(clientProperties));
```

```
        return configServicePropertySourceLocator;
    }
}
```

1. 在 `resources/META-INF` 中创建一个名为 `spring.factories` 的文件，并指定您的自定义配置。

spring.factorties

```
org.springframework.cloud.bootstrap.BootstrapConfiguration =
com.my.config.client.CustomConfigServiceBootstrapConfiguration
```

Vault

当使用 Vault 作为配置服务器的后端时，客户端将需要为服务器提供一个令牌，以从 Vault 中检索值。可以通过在 `bootstrap.yml` 中设置 `spring.cloud.config.token` 在客户端中提供此令牌。

bootstrap.yml

```
spring:
  cloud:
    config:
      token: YourVaultToken
```

Vault

Vault 中的嵌套密钥

Vault 支持将键嵌入存储在 Vault 中的值。例如

```
echo -n '{"appA": {"secret": "appAsecret"}, "bar": "baz"}' |
vault write secret/myapp -
```

此命令将向您的 Vault 编写一个 JSON 对象。要在 Spring 中访问这些值，您将使用传统的点（。）注释。例如

```
@Value("${appA.secret}")
String name = "World";
```

上述代码将 `name` 变量设置为 `appAsecret`。

Spring Cloud Netflix

Dalston.RELEASE

该项目通过自动配置为 Spring Boot 应用程序提供 Netflix OSS 集成，并绑定到 Spring 环境和其他 Spring 编程模型成语。通过几个简单的注释，您可以快速启用和配置应用程序中的常见模式，并通过经过测试的 Netflix 组件构建大型分布式系统。提供的模式包括服务发现（Eureka），断路器（Hystrix），智能路由（Zuul）和客户端负载平衡（Ribbon）。

服务发现：Eureka 客户端

服务发现是基于微服务架构的关键原则之一。尝试配置每个客户端或某种形式的约定可能非常困难，可以非常脆弱。Netflix 服务发现服务器和客户端是 Eureka。可以将服务器配置和部署为高可用性，每个服务器将注册服务的状态复制到其他服务器。

如何包含 Eureka 客户端

要在您的项目中包含 Eureka 客户端，请使用组 `org.springframework.cloud` 和工件 ID `spring-cloud-starter-eureka`

的启动器。有关 使用当前的 Spring Cloud 发布列表设置构建系统的详细信息，请参阅 Spring Cloud 项目页面。

注册 Eureka

当客户端注册 Eureka 时，它提供关于自身的元数据，例如主机和端口，健康指示符 URL，主页等。Eureka 从属于服务的每个实例接收心跳消息。如果心跳失败超过可配置的时间表，则通常将该实例从注册表中删除。

示例 eureka 客户端：

```
@Configuration
@ComponentScan
@EnableAutoConfiguration
@EnableEurekaClient
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello world";
    }

    public static void main(String[] args) {
        new
SpringApplicationBuilder(Application.class).web(true).run
(args);
    }

}
```

（即完全正常的 Spring Boot 应用程序）。在这个例子中，我们明确地使用 `@EnableEurekaClient`，但只有 Eureka 可用，你也可以使用 `@EnableDiscoveryClient`。需要配置才能找到 Eureka 服务器。例：

application.yml
```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

其中“defaultZone”是一个魔术字符串后备值，为任何不表示首选项的客户端提供服务 URL（即它是有用的默认值）。

从 `Environment` 获取的默认应用程序名称（服务 ID），虚拟主机和非安全端口分别为`${spring.application.name}`，`${spring.application.name}`和`${server.port}`。

`@EnableEurekaClient` 将应用程序同时进入一个 Eureka“实例”（即注册自己）和一个“客户端”（即它可以查询注册表以查找其他服务）。实例行为由 `eureka.instance.*`配置键驱动，但是如果您确保您的应用程序具有 `spring.application.name`（这是 Eureka 服务 ID 或 VIP 的默认值），那么默认值将是正常的。

有关可配置选项的更多详细信息，请参阅 EurekaInstanceConfigBean 和 EurekaClientConfigBean。

使用 Eureka 服务器进行身份验证

如果其中一个 `eureka.client.serviceUrl.defaultZone` 网址中包含一个凭据（如 http://user:password@localhost:8761/eureka）），HTTP 基本身份验证将自动添加到您的 eureka 客户端。对于更复杂的需求，您可以创

建 `DiscoveryClientOptionalArgs` 类型的`@Bean`，并将 `ClientFilter` 实例注入到其中，所有这些都将应用于从客户端到服务器的调用。

注意	由于 Eureka 中的限制，不可能支持每个服务器的基本身份验证凭据，所以只能使用

状态页和健康指标

Eureka 实例的状态页面和运行状况指示器分别默认为“/ info”和“/ health”，它们是 Spring Boot 执行器应用程序中有用端点的默认位置。如果您使用非默认上下文路径或 servlet 路径（例如 `server.servletPath=/foo`）或管理端点路径（例如 `management.contextPath=/admin`），则需要更改这些，即使是执行器应用程序。例：

application.yml

```
eureka:
  instance:
    statusPageUrlPath: ${management.context-path}/info
    healthCheckUrlPath: ${management.context-path}/health
```

这些链接显示在客户端使用的元数据中，并在某些情况下用于决定是否将请求发送到应用程序，因此如果它们是准确的，这是有帮助的。

注册安全应用程序

如果您的应用程序想通过 HTTPS 联系，则可以分别在

`EurekaInstanceConfig`,

即 `eureka.instance.[nonSecurePortEnabled,securePortEnabled]=[`

`false,true]` 中设置两个标志。这将使 Eureka 发布实例信息显示安全通信的明确偏好。Spring Cloud `DiscoveryClient` 将始终为以这种方式配置的服务返回一个 `https://…;` URI，并且 Eureka（本机）实例信息将具有安全的健康检查 URL。

由于 Eureka 内部的工作方式，它仍然会发布状态和主页的非安全网址，除非您也明确地覆盖。您可以使用占位符来配置 eureka 实例 URL，例如

application.yml
```
eureka:
  instance:
    statusPageUrl: https://${eureka.hostname}/info
    healthCheckUrl: https://${eureka.hostname}/health
    homePageUrl: https://${eureka.hostname}/
```

（请注意，`${eureka.hostname}`是仅在稍后版本的 Eureka 中可用的本地占位符，您也可以使用 Spring 占位符实现同样的功能，例如使用 `${eureka.instance.hostName}`。

注意	如果您的应用程序在代理服务器后面运行，并且 SSL 终止服务在代理中（例如，如果 作为服务），则需要确保代理“转发”头部被截取并处理应用程序。Spring Boot 应 执行“X-Forwarded - \ *”标头的显式配置。你这个错误的一个迹象就是你的应用 的主机，端口或协议）。

Eureka 的健康检查

默认情况下，Eureka 使用客户端心跳来确定客户端是否启动。除非另有规定，否则发现客户端将不会根据 Spring Boot 执行器传播应用程序的当前运行状况检查状态。这意味着成功注册后 Eureka 将永远宣布申请处于“UP”状态。通过启用

Eureka 运行状况检查可以改变此行为，从而将应用程序状态传播到 Eureka。因此，每个其他应用程序将不会在“UP”之外的状态下将流量发送到应用程序。

application.yml
```
eureka:
  client:
    healthcheck:
      enabled: true
```

警告	`eureka.client.healthcheck.enabled=true` 只能在 `application.yml` 中设 导致不期望的副作用，例如在具有 `UNKNOWN` 状态的 eureka 中注册。

如果您需要更多的控制健康检查，您可以考虑实施自己的 `com.netflix.appinfo.HealthCheckHandler`。

Eureka 实例和客户端的元数据

值得花点时间了解 Eureka 元数据的工作原理，以便您可以在平台上使用它。有主机名，IP 地址，端口号，状态页和运行状况检查等标准元数据。这些发布在服务注册表中，由客户使用，以直接的方式联系服务。额外的元数据可以添加到 `eureka.instance.metadataMap` 中的实例注册中，并且这将在远程客户端中可访问，但一般不会更改客户端的行为，除非意识到元数据的含义。下面描述了几个特殊情况，其中 Spring Cloud 已经为元数据映射指定了含义。

在 Cloudfoundry 上使用 Eureka

Cloudfoundry 有一个全局路由器，所以同一个应用程序的所有实例都具有相同的主机名（在具有相似架构的其他 PaaS 解决方案中也是如此）。这不一定是使用 Eureka 的障碍，但如果您使用路由器（建议，甚至是强制性的，具体取决于您

的平台的设置方式），则需要明确设置主机名和端口号（安全或非安全），以便他们使用路由器。您可能还需要使用实例元数据，以便您可以区分客户端上的实例（例如，在自定义负载平衡器中）。默认情况下，`eureka.instance.instanceId`为`vcap.application.instance_id`。例如：

application.yml

```
eureka:
  instance:
    hostname: ${vcap.application.uris[0]}
    nonSecurePort: 80
```

根据 Cloudfoundry 实例中安全规则的设置方式，您可以注册并使用主机 VM 的 IP 地址进行直接的服务到服务调用。此功能尚未在 Pivotal Web Services（PWS）上提供。

在 AWS 上使用 Eureka

如果应用程序计划将部署到 AWS 云，那么 Eureka 实例必须被配置为 AWS 意识到，这可以通过定制来完成 EurekaInstanceConfigBean 方式如下：

```
@Bean
@Profile("!default")
public EurekaInstanceConfigBean
eurekaInstanceConfig(InetUtils inetUtils) {
  EurekaInstanceConfigBean b = new
EurekaInstanceConfigBean(inetUtils);
  AmazonInfo info =
AmazonInfo.Builder.newBuilder().autoBuild("eureka");
  b.setDataCenterInfo(info);
  return b;
}
```

更改 Eureka 实例 ID

香草 Netflix Eureka 实例注册了与其主机名相同的 ID（即每个主机只有一个服务）。Spring Cloud Eureka 提供了一个明智的默认，如下所示：`${spring.cloud.client.hostname}:${spring.application.name}:${spring.application.instance_id:${server.port}}}`。例如 `myhost:myappname:8080`。

使用 Spring Cloud，您可以通过在 `eureka.instance.instanceId` 中提供唯一的标识符来覆盖此。例如：

application.yml

```
eureka:
  instance:
    instanceId:
${spring.application.name}:${vcap.application.instance_id
:${spring.application.instance_id:${random.value}}}
```

使用这个元数据和在 localhost 上部署的多个服务实例，随机值将在那里进行，以使实例是唯一的。在 Cloudfoundry 中，`vcap.application.instance_id` 将在 Spring Boot 应用程序中自动填充，因此不需要随机值。

使用 EurekaClient

一旦您拥有`@EnableDiscoveryClient`（或`@EnableEurekaClient`）的应用程序，您就可以使用它来从 Eureka 服务器发现服务实例。一种方法是使用本机 `com.netflix.discovery.EurekaClient`（而不是 Spring 云 `DiscoveryClient`），例如

```
@Autowired
private EurekaClient discoveryClient;

public String serviceUrl() {
    InstanceInfo instance =
discoveryClient.getNextServerFromEureka("STORES", false);
    return instance.getHomePageUrl();
}
```

提示	不要使用@PostConstruct 方法或@Scheduled 方法（或 ApplicationContext EurekaClient。它被初始化为 SmartLifecycle（带有 phase=0），所以最早可 段的 SmartLifecycle。

本机 Netflix EurekaClient 的替代方案

您不必使用原始的 Netflix EurekaClient，通常在某种包装器后面使用它更为方便。Spring Cloud 支持 Feign（REST 客户端构建器），还支持 Spring RestTemplate 使用逻辑 Eureka 服务标识符（VIP）而不是物理 URL。要使用固定的物理服务器列表配置 Ribbon，您可以将 <client>.ribbon.listOfServers 设置为逗号分隔的物理地址（或主机名）列表，其中<client>是客户端的 ID。

您还可以使用 org.springframework.cloud.client.discovery.DiscoveryClient，它为 Netflix 不具体的发现客户端提供简单的 API，例如

```
@Autowired
private DiscoveryClient discoveryClient;

public String serviceUrl() {
    List<ServiceInstance> list =
discoveryClient.getInstances("STORES");
    if (list != null && list.size() > 0 ) {
        return list.get(0).getUri();
```

```
    }
    return null;
}
```

为什么注册服务这么慢?

作为一个实例也包括定期心跳到注册表（通过客户端的 `serviceUrl`），默认持续时间为 30 秒。在实例，服务器和客户端在其本地缓存中都具有相同的元数据（因此可能需要 3 个心跳）之前，客户端才能发现服务。您可以使用 `eureka.instance.leaseRenewalIntervalInSeconds` 更改期限，这将加快客户端连接到其他服务的过程。在生产中，最好坚持使用默认值，因为服务器内部有一些计算可以对租赁更新期进行假设。



区

如果您已将 Eureka 客户端部署到多个区域，您可能希望这些客户端在使用另一个区域中的服务之前，利用同一区域内的服务。为此，您需要正确配置您的 Eureka 客户端。

首先，您需要确保将 Eureka 服务器部署到每个区域，并且它们是彼此的对等体。有关详细信息，请参阅区域和区域部分 。

接下来，您需要告知 Eureka 您的服务所在的区域。您可以使用 `metadataMap` 属性来执行此操作。例如，如果 `service 1` 部署到 `zone 1` 和 `zone 2`，则需要在 `service 1` 中设置以下 Eureka 属性

1 区服务 1

```
eureka.instance.metadataMap.zone = zone1
eureka.client.preferSameZoneEureka = true
```

第 2 区的服务 1

```
eureka.instance.metadataMap.zone = zone2
eureka.client.preferSameZoneEureka = true
```

服务发现：Eureka 服务器

如何包含 Eureka 服务器

要在项目中包含 Eureka 服务器，请使用组 `org.springframework.cloud` 和工件 id `spring-cloud-starter-eureka-server` 的启动器。有关 使用当前的 Spring Cloud 发布列表设置构建系统的详细信息，请参阅 Spring Cloud 项目页面。

如何运行 Eureka 服务器

示例 eureka 服务器;

```
@SpringBootApplication
@EnableEurekaServer
public class Application {

    public static void main(String[] args) {
        new
SpringApplicationBuilder(Application.class).web(true).run
(args);
```

```
    }
}
```

服务器具有一个带有 UI 的主页，并且根据`/eureka/*`下的正常 Eureka 功能的 HTTP API 端点。

Eureka 背景阅读：看助焊剂电容和谷歌小组讨论。

提示

由于 Gradle 的依赖关系解决规则和父母的 bom 功能缺乏，只要依靠 spring-cloud-st
序启动失败。要解决这个问题，必须添加 Spring Boot Gradle 插件，并且必须导入 S

的 build.gradle

```
buildscript {
  dependencies {
    classpath("org.springframework.boot:spring-boot-gradle-pl
  }
}

apply plugin: "spring-boot"

dependencyManagement {
  imports {
    mavenBom "org.springframework.cloud:spring-cloud-dependen
  }
}
```

高可用性，区域和地区

Eureka 服务器没有后端存储，但是注册表中的服务实例都必须发送心跳以保持其注册更新（因此可以在内存中完成）。客户端还具有 eureka 注册的内存缓存（因此，他们不必为注册表提供每个服务请求）。

默认情况下，每个 Eureka 服务器也是一个 Eureka 客户端，并且需要（至少一个）服务 URL 来定位对等体。如果您不提供该服务将运行和工作，但它将淋浴您的日志与大量的噪音无法注册对等体。

关于区域和区域的客户端 Ribbon 支持的详细信息，请参见下文。

独立模式

只要存在某种监视器或弹性运行时间（例如 Cloud Foundry），两个高速缓存（客户机和服务器）和心跳的组合使独立的 Eureka 服务器对故障具有相当的弹性。在独立模式下，您可能更喜欢关闭客户端行为，因此不会继续尝试并且无法访问其对等体。例：

application.yml （Standalone Eureka Server）

```
server:
  port: 8761

eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
      defaultZone:
http://${eureka.instance.hostname}:${server.port}/eureka/
```

请注意，`serviceUrl` 指向与本地实例相同的主机。

同行意识

通过运行多个实例并请求他们相互注册，可以使 Eureka 更具弹性和可用性。事实上，这是默认的行为，所以你需要做的只是为对方添加一个有效的 `serviceUrl`，例如

application.yml（Two Peer Aware Eureka 服务器）

```
---
spring:
  profiles: peer1
eureka:
  instance:
    hostname: peer1
  client:
    serviceUrl:
      defaultZone: http://peer2/eureka/

---
spring:
  profiles: peer2
eureka:
  instance:
    hostname: peer2
  client:
    serviceUrl:
      defaultZone: http://peer1/eureka/
```

在这个例子中，我们有一个 YAML 文件，可以通过在不同的 Spring 配置文件中运行，在 2 台主机（peer1 和 peer2）上运行相同的服务器。您可以使用此配置来测试单个主机上的对等体感知（通过操作`/etc/hosts`来解析主机名，在生产中没有太多价值）。事实上，如果您在一台知道自己的主机名的机器上运行（默认情况下使用`java.net.InetAddress`查找），则不需要 `eureka.instance.hostname`。

您可以向系统添加多个对等体，只要它们至少一个边缘彼此连接，则它们将在它们之间同步注册。如果对等体在物理上分离（在数据中心内或多个数据中心之间），则系统原则上可以分裂脑型故障。

喜欢 IP 地址

在某些情况下，Eureka 优先发布服务的 IP 地址而不是主机名。将 `eureka.instance.preferIpAddress` 设置为 `true`，并且当应用程序向 eureka 注册时，它将使用其 IP 地址而不是其主机名。

断路器：Hystrix 客户端

Netflix 的创造了一个调用的库 Hystrix 实现了断路器图案。在微服务架构中，通常有多层服务调用。

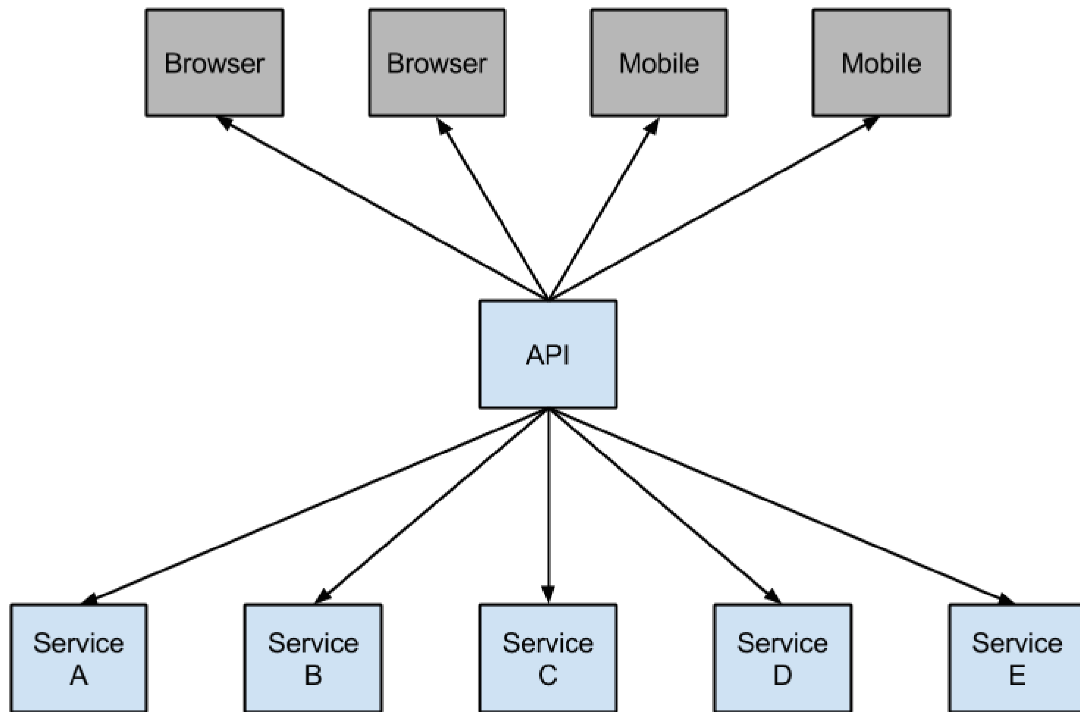


图 1.微服务图

较低级别的服务中的服务故障可能导致用户级联故障。当对特定服务的呼叫达到一定阈值时（Hystrix 中的默认值为 5 秒内的 20 次故障），电路打开，不进行通话。在错误和开路的情况下，开发人员可以提供后备。

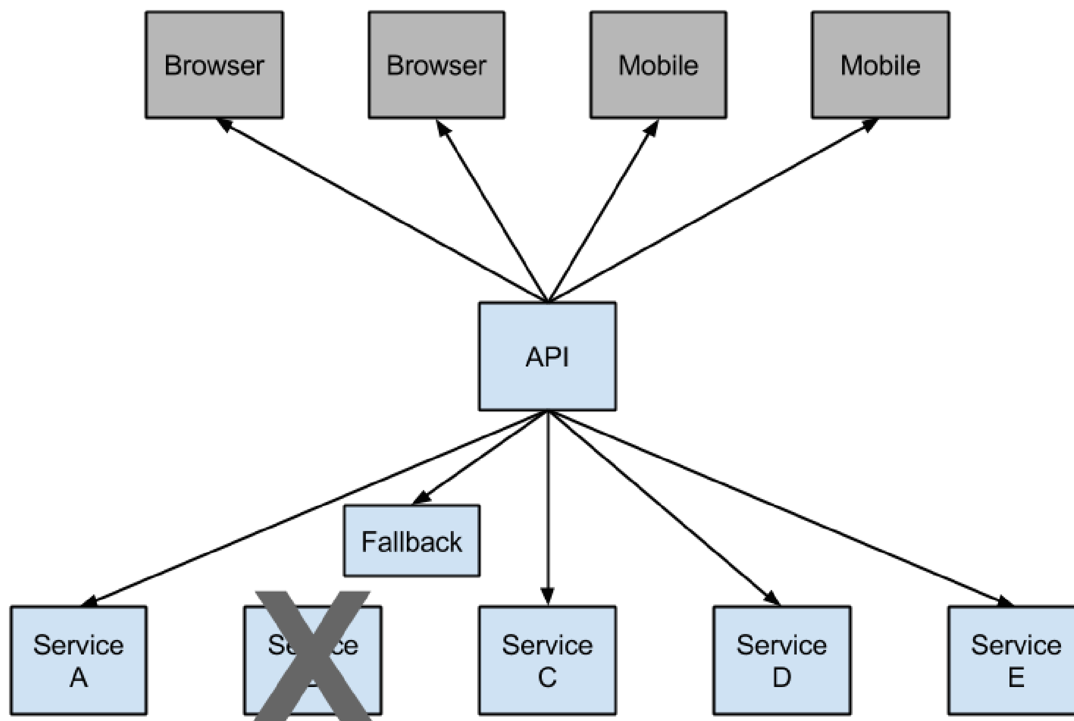


图 2. Hystrix 回退防止级联故障

开放式电路会停止级联故障，并允许不必要的或失败的服务时间来愈合。回退可以是另一个 Hystrix 保护的调用，静态数据或一个正常的空值。回退可能被链接，所以第一个回退使得一些其他业务电话又回到静态数据。

如何加入 Hystrix

要在项目中包含 Hystrix，请使用组 `org.springframework.cloud` 和 artifact id `spring-cloud-starter-hystrix` 的启动器。有关 使用当前的 Spring Cloud 发布列表设置构建系统的详细信息，请参阅 [Spring Cloud 项目页面](#)。

示例启动应用程序:

```
@SpringBootApplication
@EnableCircuitBreaker
public class Application {
```

```
    public static void main(String[] args) {
        new
SpringApplicationBuilder(Application.class).web(true).run
(args);
    }

}

@Component
public class StoreIntegration {

    @HystrixCommand(fallbackMethod = "defaultStores")
    public Object getStores(Map<String, Object>
parameters) {
        //do stuff that might fail
    }

    public Object defaultStores(Map<String, Object>
parameters) {
        return /* something useful */;
    }
}
```

`@HystrixCommand`由名为"javanica"的 Netflix contrib 库提供。Spring Cloud 在连接到 Hystrix 断路器的代理中使用该注释自动包装 Spring bean。断路器计算何时打开和关闭电路，以及在发生故障时应该做什么。

要配置`@HystrixCommand`，您可以使用 `commandProperties` 属性列出 `@HystrixProperty` 注释。请参阅 这里 了解更多详情。有关 可用属性的详细信息，请参阅 Hystrix 维基。

传播安全上下文或使用 Spring 范围

如果您希望某些线程本地上下文传播到`@HystrixCommand`，默认声明将不起作用，因为它在线程池中执行命令（超时）。您可以使用某些配置或直接在注释中使用与使用相同的线程来调用 Hystrix，方法是要求使用不同的“隔离策略”。例如：

```
@HystrixCommand(fallbackMethod = "stubMyService",
    commandProperties = {

@HystrixProperty(name="execution.isolation.strategy",
value="SEMAPHORE")
    }
)
...
```

如果您使用`@SessionScope`或`@RequestScope`，同样的事情也适用。您将知道何时需要执行此操作，因为运行时异常说它找不到范围的上下文。

您还可以将 `hystrix.shareSecurityContext` 属性设置为 `true`。这样做会自动配置一个 Hystrix 并发策略插件钩子，他将 `SecurityContext` 从主线程传送到 Hystrix 命令使用的钩子。Hystrix 不允许注册多个 hystrix 并发策略，因此可以通过将自己的 `HystrixConcurrencyStrategy` 声明为 Spring bean 来实现扩展机制。Spring Cloud 将在 Spring 上下文中查找您的实现，并将其包装在自己的插件中。

健康指标

连接断路器的状态也暴露在呼叫应用程序的`/health`端点中。

```
{
    "hystrix": {
```

```
        "openCircuitBreakers": [
            "StoreIntegration::getStoresByLocationLink"
        ],
        "status": "CIRCUIT_OPEN"
    },
    "status": "UP"
}
```

Hystrix 指标流

要使 Hystrix 指标流包含对 `spring-boot-starter-actuator` 的依赖。这将使`/hystrix.stream` 作为管理端点。

```
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-
actuator</artifactId>
    </dependency>
```

断路器：Hystrix 仪表板

Hystrix 的主要优点之一是它收集关于每个 HystrixCommand 的一套指标。

Hystrix 仪表板以有效的方式显示每个断路器的运行状况。

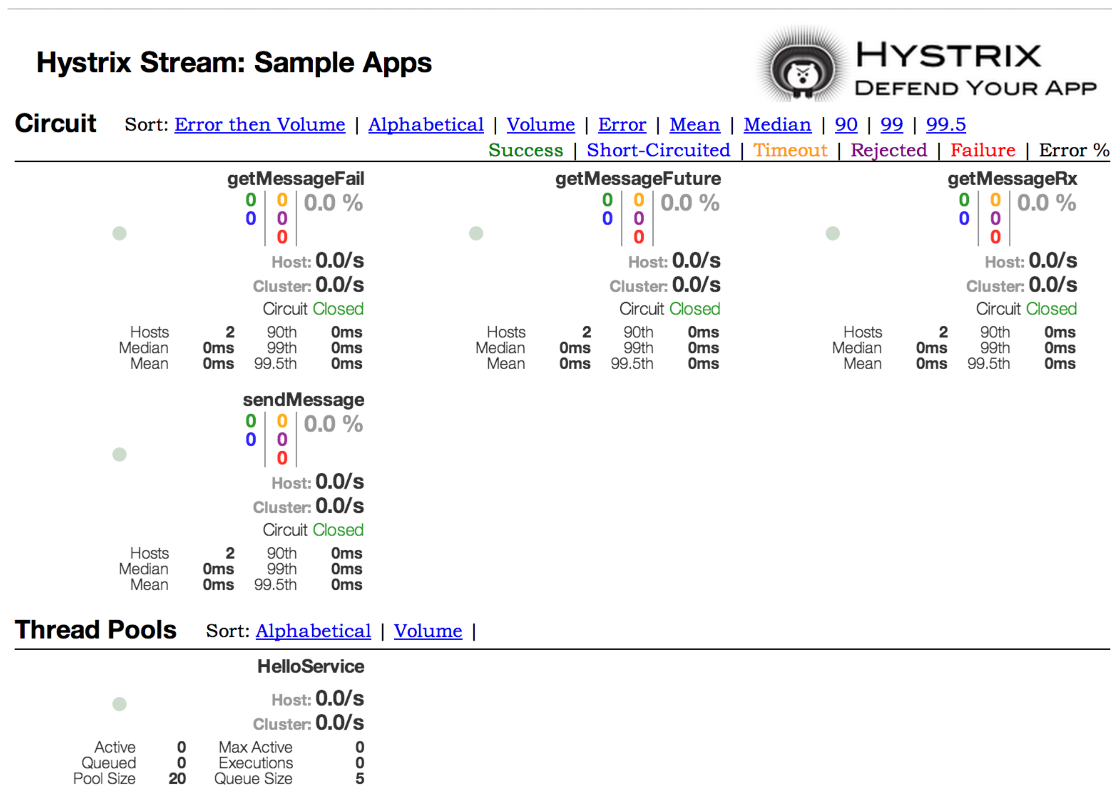


图3. Hystrix 仪表板

Hystrix 超时和 Ribbon 客户

当使用包含 Ribbon 客户端的 Hystrix 命令时，您需要确保您的 Hystrix 超时配置为长于配置的 Ribbon 超时，包括可能进行的任何潜在的重试。例如，如果您的 Ribbon 连接超时为一秒钟，并且 Ribbon 客户端可能会重试该请求三次，那么您的 Hystrix 超时应该略超过三秒钟。

如何包含 Hystrix 仪表板

要在项目中包含 Hystrix 仪表板，请使用组 `org.springframework.cloud` 和工件 ID `spring-cloud-starter-hystrix-dashboard` 的启动器。有关 使用

当前的 Spring Cloud 发布列表设置构建系统的详细信息，请参阅 Spring Cloud 项目页面。

要运行 Hystrix 仪表板使用`@EnableHystrixDashboard`注释您的 Spring Boot 主类。然后访问`/hystrix`，并将仪表板指向 Hystrix 客户端应用程序中的单个实例`/hystrix.stream`端点。

Turbine

从个人实例看，Hystrix 数据在系统整体健康方面不是非常有用。Turbine 是将所有相关`/hystrix.stream`端点聚合到 Hystrix 仪表板中使用的`/turbine.stream`的应用程序。个人实例位于 Eureka。运行 Turbine 就像使用`@EnableTurbine`注释（例如使用 spring-cloud-starter-turbine 设置类路径）注释主类一样简单。来自 Turbine 1 维基的所有文档配置属性都适用。唯一的区别是`turbine.instanceUrlSuffix`不需要预先添加的端口，除非`turbine.instanceInsertPort=false`自动处理。

注意	默认情况下，Turbine 通过在 Eureka 中查找其`homePageUrl`条目，然后将`/hyst` `/hystrix.stream`端点。这意味着如果`spring-boot-actuator`在自己的端口 `/hystrix.stream`的调用将失败。要使涡轮机找到正确端口的 Hystrix 流，您需 `management.port`:

```
eureka:
  instance:
    metadata-map:
      management.port: ${management.port:8081}
```

配置密钥`turbine.appConfig`是涡轮机将用于查找实例的尤里卡服务列表。

涡轮流然后在 Hystrix 仪表板中使用如下 URL：

http://my.turbine.sever:8080/turbine.stream?cluster=<CLUSTER

NAME>；（如果名称为“默认值”，则可以省略群集参数）。`cluster` 参数必须与 `turbine.aggregator.clusterConfig` 中的条目相匹配。从 eureka 返回的值是大写字母，因此如果有一个名为“customers”的 Eureka 注册了一个应用程序，我们预计此示例可以正常工作：

```
turbine:
  aggregator:
    clusterConfig: CUSTOMERS
  appConfig: customers
```

`clusterName` 可以通过 `turbine.clusterNameExpression` 中的 SPEL 表达式以 root 身份 `InstanceInfo` 进行自定义。默认值为 `appName`，这意味着 Eureka serviceId 最终作为集群密钥（即客户的 `InstanceInfo` 具有 `appName`“CUSTOMERS”）。一个不同的例子是 `turbine.clusterNameExpression=aSGName`，它将从 AWS ASG 名称获取集群名称。另一个例子：

```
turbine:
  aggregator:
    clusterConfig: SYSTEM,USER
  appConfig: customers,stores,ui,admin
  clusterNameExpression: metadata['cluster']
```

在这种情况下，来自 4 个服务的集群名称从其元数据映射中提取，并且预期具有包含“SYSTEM”和“USER”的值。

要为所有应用程序使用“默认”集群，您需要一个字符串文字表达式（带单引号，并且如果它在 YAML 中也使用双引号进行转义）：

```
turbine:
  appConfig: customers,stores
```

```
  clusterNameExpression: "'default'"
```

Spring Cloud 提供了一个 `spring-cloud-starter-turbine`，它具有运行 Turbine 服务器所需的所有依赖关系。只需创建一个 Spring Boot 应用程序并用 `@EnableTurbine` 注释它。

注意	默认情况下，Spring Cloud 允许 Turbine 使用主机和端口允许每个主机在每个群集 Turbine 本地 Netflix 的行为，它不会允许每个主机上的多个过程，每簇（关键实例 `turbine.combineHostPort=false`。

Turbine Stream

在某些环境中（例如，在 PaaS 设置中），从所有分布式 Hystrix 命令中提取度量的经典 Turbine 模型不起作用。在这种情况下，您可能希望让 Hystrix 命令将度量标准推送到 Turbine，并且 Spring Cloud 可以使用消息传递。您需要在客户端上执行的所有操作都为您选择的 `spring-cloud-netflix-hystrix-stream` 和 `spring-cloud-starter-stream-*`添加依赖关系（有关经纪人的详细信息，请参阅 Spring Cloud Stream 文档，以及如何配置客户端凭据，但是应该为当地经纪人开箱即用）。

在服务器端只需创建一个 Spring Boot 应用程序并使用 `@EnableTurbineStream` 进行注释，默认情况下将在 8989 端口（将您的 Hystrix 仪表板指向该端口，任何路径）。您可以使用 `server.port` 或 `turbine.stream.port` 自定义端口。如果类路径中还有 `spring-boot-starter-web` 和 `spring-boot-starter-actuator`，那么您可以通过提供

不同的`management.port`在单独端口（默认情况下使用 Tomcat）打开 Actuator 端点。

然后，您可以将 Hystrix 仪表板指向 Turbine Stream 服务器，而不是单个 Hystrix 流。如果 Turbine Stream 在 myhost 上的端口 8989 上运行，则将 http://myhost:8989 放在 Hystrix 仪表板中的流输入字段中。电路将以各自的 serviceId 为前缀，后跟一个点，然后是电路名称。

Spring Cloud 提供了一个`spring-cloud-starter-turbine-stream`，它具有您需要的 Turbine Stream 服务器运行所需的所有依赖项，只需添加您选择的 Stream binder，例如`spring-cloud-starter-stream-rabbit`。您需要 Java 8 来运行应用程序，因为它是基于 Netty 的。

客户端负载平衡器：Ribbon

Ribbon 是一个客户端负载均衡器，它可以很好地控制 HTTP 和 TCP 客户端的行为。Feign 已经使用 Ribbon，所以如果您使用`@FeignClient`，则本节也适用。

Ribbon 中的中心概念是指定客户端的概念。每个负载平衡器是组合的组合的一部分，它们一起工作以根据需要联系远程服务器，并且集合具有您将其作为应用程序开发人员（例如使用`@FeignClient`注释）的名称。Spring Cloud 使用`RibbonClientConfiguration`为每个命名的客户端根据需要创建一个新的合奏作为`ApplicationContext`。这包含（除其他外）`ILoadBalancer`，`RestClient`和`ServerListFilter`。

如何加入 Ribbon

要在项目中包含 Ribbon，请使用组 org.springframework.cloud 和工件 ID spring-cloud-starter-ribbon 的起始器。有关使用当前的 Spring Cloud 发布列表设置构建系统的详细信息，请参阅 Spring Cloud 项目页面。

自定义 Ribbon 客户端

您可以使用<client>.ribbon.*中的外部属性来配置 Ribbon 客户端的某些位，这与使用 Netflix API 本身没有什么不同，只能使用 Spring Boot 配置文件。本机选项可以在 CommonClientConfigKey（功能区内核心部分）中作为静态字段进行检查。

Spring Cloud 还允许您通过使用@RibbonClient 声明其他配置（位于 RibbonClientConfiguration 之上）来完全控制客户端。例：

```
@Configuration
@RibbonClient(name = "foo", configuration =
FooConfiguration.class)
public class TestConfiguration {
}
```

在这种情况下，客户端由 RibbonClientConfiguration 中已经存在的组件与 FooConfiguration 中的任何组件组成（后者通常会覆盖前者）。

警告	FooConfiguration 必须是@Configuration，但请注意，它不在主应用程序上下 所有@RibbonClients 共享。如果您使用@ComponentScan（或@SpringBootAp 含（例如将其放在一个单独的，不重叠的包中，或者指定要在@ComponentScan）

Spring Cloud Netflix 默认情况下为 Ribbon（BeanType beanName：ClassName）提供以下 bean：

- IClientConfig ribbonClientConfig：DefaultClientConfigImpl
- IRule ribbonRule：ZoneAvoidanceRule
- IPing ribbonPing：NoOpPing
- ServerList<Server> ribbonServerList：ConfigurationBasedServerList
- ServerListFilter<Server> ribbonServerListFilter：ZonePreferenceServerListFilter
- ILoadBalancer ribbonLoadBalancer：ZoneAwareLoadBalancer
- ServerListUpdater ribbonServerListUpdater：PollingServerListUpdater

创建一个类型的 bean 并将其放置在@RibbonClient 配置（例如上面的 FooConfiguration）中）允许您覆盖所描述的每个 bean。例：

```
@Configuration
public class FooConfiguration {
    @Bean
    public IPing ribbonPing(IClientConfig config) {
        return new PingUrl();
    }
}
```

这用 PingUrl 代替 NoOpPing。

使用属性自定义 Ribbon 客户端

从版本 1.2.0 开始，Spring Cloud Netflix 现在支持使用属性与 Ribbon 文档兼容来自定义 Ribbon 客户端。

这允许您在不同环境中更改启动时的行为。

支持的属性如下所示，应以`<clientName>.ribbon.`为前缀：

- `NFLoadBalancerClassName`：应实施 `ILoadBalancer`
- `NFLoadBalancerRuleClassName`：应实施 `IRule`
- `NFLoadBalancerPingClassName`：应实施 `IPing`
- `NIWSServerListClassName`：应实施 `ServerList`
- `NIWSServerListFilterClassName` 应实施 `ServerListFilter`

注意	在这些属性中定义的类优先于使用`@RibbonClient(configuration=MyRibbon` Spring Cloud Netflix 提供的默认值。

要设置服务名称 `users` 的 `IRule`，您可以设置以下内容：

application.yml

```
users:
  ribbon:
    NFLoadBalancerRuleClassName:
com.netflix.loadbalancer.WeightedResponseTimeRule
```

有关 Ribbon 提供的实现，请参阅 Ribbon 文档。

在 Eureka 中使用 Ribbon

当 Eureka 与 Ribbon 结合使用（即两者都在类路径上）时，`ribbonServerList` 将被扩展为 `DiscoveryEnabledNIWSServerList`，扩展名为 Eureka 的服务器列表。它还用 `NIWSDiscoveryPing` 替换 `IPing` 接口，代理到 Eureka 以确定服务器是否启动。默认情况下安装的 `ServerList` 是一个 `DomainExtractingServerList`，其目的是使物理元数据可用于负载平衡器，而不使用 AWS AMI 元数据（这是 Netflix 依赖的）。默认情况下，服务器列表将使用实例元数据（如远程客户端集合 `eureka.instance.metadataMap.zone`）中提供的“区域”信息构建，如果缺少，则可以使用服务器主机名中的域名作为代理用于区域（如果设置了标志 `approximateZoneFromHostname`）。一旦区域信息可用，它可以在 `ServerListFilter` 中使用。默认情况下，它将用于定位与客户端相同区域的服务器，因为默认值为 `ZonePreferenceServerListFilter`。默认情况下，客户端的区域与远程实例的方式相同，即通过 `eureka.instance.metadataMap.zone`。

注意	设置客户端区域的正统“archaius”方式是通过一个名为“@zone”的配置属性，如 有其他设置（请注意，该键必须被引用）在 YAML 配置中）。
注意	如果没有其他的区域数据源，则基于客户端配置（与实例配置相反）进行猜测。我们 `eureka.client.availabilityZones`（从区域名称映射到区域列表），并将实 `eureka.client.region`，其默认为“us-east-1”为与本机 Netflix 的兼容性）

示例：如何使用 Ribbon 不使用 Eureka

Eureka 是一种方便的方式来抽象远程服务器的发现，因此您不必在客户端中对其 URL 进行硬编码，但如果您不想使用它，Ribbon 和 Feign 仍然是适用的。假

设您已经为“商店”申请了`@RibbonClient`，并且 Eureka 未被使用（甚至不在类路径上）。Ribbon 客户端默认为已配置的服务器列表，您可以提供这样的配置

application.yml

```
stores:
  ribbon:
    listOfServers: example.com,google.com
```

示例：在 Ribbon 中禁用 Eureka 使用

设置属性`ribbon.eureka.enabled = false`将明确禁用在 Ribbon 中使用 Eureka。

application.yml

```
ribbon:
  eureka:
   enabled: false
```

直接使用 Ribbon API

您也可以直接使用`LoadBalancerClient`。例：

```
public class MyClass {
    @Autowired
    private LoadBalancerClient loadBalancer;

    public void doStuff() {
        ServiceInstance instance =
loadBalancer.choose("stores");
        URI storesUri =
URI.create(String.format("http://%s:%s",
instance.getHost(), instance.getPort()));
        // ... do something with the URI
    }
}
```

缓存 Ribbon 配置

每个 Ribbon 命名的客户端都有一个相应的子应用程序上下文，Spring Cloud 维护，这个应用程序上下文在第一个请求中被延迟加载到命名的客户端。可以通过指定 Ribbon 客户端的名称，在启动时，可以更改此延迟加载行为，从而热切加载这些子应用程序上下文。

application.yml

```
ribbon:
  eager-load:
    enabled: true
    clients: client1, client2, client3
```

声明性 REST 客户端：Feign

Feign 是一个声明式的 Web 服务客户端。这使得 Web 服务客户端的写入更加方便 要使用 Feign 创建一个界面并对其进行注释。它具有可插入注释支持，包括 Feign 注释和 JAX-RS 注释。Feign 还支持可插拔编码器和解码器。Spring Cloud 增加了对 Spring MVC 注释的支持，并使用 Spring Web 中默认使用的 `HttpMessageConverters`。Spring Cloud 集成 Ribbon 和 Eureka 以在使用 Feign 时提供负载均衡的 http 客户端。

如何加入 Feign

要在您的项目中包含 Feign，请使用组 `org.springframework.cloud` 和工件 ID `spring-cloud-starter-feign` 的启动器。有关 使用当前的 Spring Cloud 发布列表设置构建系统的详细信息，请参阅 Spring Cloud 项目页面。

示例 spring boot 应用

```
@Configuration
@ComponentScan
@EnableAutoConfiguration
@EnableEurekaClient
@EnableFeignClients
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

StoreClient.java

```
@FeignClient("stores")
public interface StoreClient {
    @RequestMapping(method = RequestMethod.GET, value =
"/stores")
    List<Store> getStores();

    @RequestMapping(method = RequestMethod.POST, value =
"/stores/{storeId}", consumes = "application/json")
    Store update(@PathVariable("storeId") Long storeId,
Store store);
}
```

在`@FeignClient` 注释中，String 值（以上“存储”）是一个任意的客户端名称，用于创建 Ribbon 负载平衡器（有关 Ribbon 支持的详细信息，请参阅下文））。您还可以使用 `url` 属性（绝对值或只是主机名）指定 URL。应用程序上下文中的 bean 的名称是该接口的完全限定名称。要指定您自己的别名值，您可以使用 `@FeignClient` 注释的 `qualifier` 值。

以上的 Ribbon 客户端将会发现"商店"服务的物理地址。如果您的应用程序是 Eureka 客户端，那么它将解析 Eureka 服务注册表中的服务。如果您不想使用 Eureka，您可以简单地配置外部配置中的服务器列表（例如，参见 上文）。

覆盖 Feign 默认值

Spring Cloud 的 Feign 支持的中心概念是指定的客户端。每个假装客户端都是组合的组件的一部分，它们一起工作以根据需要联系远程服务器，并且该集合具有您将其作为应用程序开发人员使用`@FeignClient`注释的名称。Spring Cloud 根据需要，使用 `FeignClientsConfiguration` 为每个已命名的客户端创建一个新的集合 `ApplicationContext`。这包含（除其他外）`feign.Decoder`，`feign.Encoder` 和 `feign.Contract`。

Spring Cloud 可以通过使用`@FeignClient` 声明额外的配置（`FeignClientsConfiguration`）来完全控制假客户端。例：

```
@FeignClient(name = "stores", configuration =
FooConfiguration.class)
public interface StoreClient {
    //..
}
```

在这种情况下，客户端由 `FeignClientsConfiguration` 中的组件与 `FooConfiguration` 中的任何组件组成（后者将覆盖前者）。

注意	`FooConfiguration` 不需要使用`@Configuration` 注释。但是，如果是，则请注 除，否则将包含此配置，因为它将成为 `feign.Decoder`，`feign.Encoder`，`fei` 时。这可以通过将其放置在任何`@ComponentScan` 或`@SpringBootApplication` `@ComponentScan` 中明确排除。

注意	`serviceId`属性现在已被弃用，有利于`name`属性。
警告	以前，使用`url`属性，不需要`name`属性。现在需要使用`name`。

`name`和`url`属性支持占位符。

```
@FeignClient(name = "${feign.name}", url =
"${feign.url}")
public interface StoreClient {
    //..
}
```

Spring Cloud Netflix 默认为 feign（`BeanType` beanName：`ClassName`）提供以下 bean：

- `Decoder` feignDecoder：`ResponseEntityDecoder`（其中包含`SpringDecoder`）
- `Encoder` feignEncoder：`SpringEncoder`
- `Logger` feignLogger：`Slf4jLogger`
- `Contract` feignContract：`SpringMvcContract`
- `Feign.Builder` feignBuilder：`HystrixFeign.Builder`
- `Client` feignClient：如果 Ribbon 启用，则为`LoadBalancerFeignClient`，否则将使用默认的 feign 客户端。

可以通过将`feign.okhttp.enabled`或`feign.httpclient.enabled`设置为`true`，并将它们放在类路径上来使用 OkHttpClient 和 ApacheHttpClient feign 客户端。

Spring Cloud Netflix 默认情况下*不*提供以下 bean，但是仍然从应用程序上下文中查找这些类型的 bean 以创建假客户机：

- Logger.Level
- Retryer
- ErrorDecoder
- Request.Options
- Collection<RequestInterceptor>
- SetterFactory

创建一个类型的 bean 并将其放置在@FeignClient 配置（例如上面的 FooConfiguration）中）允许您覆盖所描述的每个 bean。例：

```
@Configuration
public class FooConfiguration {
    @Bean
    public Contract feignContract() {
        return new feign.Contract.Default();
    }

    @Bean
    public BasicAuthRequestInterceptor
basicAuthRequestInterceptor() {
        return new BasicAuthRequestInterceptor("user",
"password");
    }
}
```

这将 SpringMvcContract 替换为 feign.Contract.Default，并将 RequestInterceptor 添加到 RequestInterceptor 的集合中。

可以在@EnableFeignClients 属性 defaultConfiguration 中以与上述相似的方式指定默认配置。不同之处在于，此配置将适用于*所有*假客户端。

注意	如果您需要在 `RequestInterceptor`s you will need to either set the for Hystrix to `SEMAPHORE` 中使用 `ThreadLocal` 绑定变量，或在 Feign 中

application.yml

```
# To disable Hystrix in Feign
feign:
  hystrix:
    enabled: false

# To set thread isolation to SEMAPHORE
hystrix:
  command:
    default:
      execution:
        isolation:
          strategy: SEMAPHORE
```

手动创建 Feign 客户端

在某些情况下，可能需要以上述方法不可能自定义您的 Feign 客户端。在这种情况下，您可以使用 [Feign Builder API]创建客户端 。下面是一个创建两个具有相同接口的 Feign 客户端的示例，但是使用单独的请求拦截器配置每个客户端。

```
@Import(FeignClientsConfiguration.class)
class FooController {

	private FooClient fooClient;

	private FooClient adminClient;

    @Autowired
	public FooController(
			Decoder decoder, Encoder encoder,
Client client) {
		this.fooClient =
Feign.builder().client(client)
				.encoder(encoder)
				.decoder(decoder)
```

```
                              .requestInterceptor(new
BasicAuthRequestInterceptor("user", "user"))
                              .target(FooClient.class,
"http://PROD-SVC");
               this.adminClient =
Feign.builder().client(client)
                              .encoder(encoder)
                              .decoder(decoder)
                              .requestInterceptor(new
BasicAuthRequestInterceptor("admin", "admin"))
                              .target(FooClient.class,
"http://PROD-SVC");
    }
}
```

注意	在上面的例子中，`FeignClientsConfiguration.class` 是 Spring Cloud Netfl
注意	`PROD-SVC` 是客户端将要求的服务的名称。

Feign Hystrix 支持

如果 Hystrix 在类路径上，`feign.hystrix.enabled=true`，Feign 将用断路器包装所有方法。还可以返回 `com.netflix.hystrix.HystrixCommand`。这样就可以使用无效模式（调用`.toObservable()`或`.observe()`或异步使用（调用`.queue()`））。

要在每个客户端基础上禁用 Hystrix 支持创建一个带有“原型”范围的香草 `Feign.Builder`，例如：

```
@Configuration
public class FooConfiguration {
    @Bean
       @Scope("prototype")
       public Feign.Builder feignBuilder() {
              return Feign.builder();
       }
}
```

警告	在 Spring Cloud 达尔斯顿发布之前，如果 Hystrix 在类路径 Feign 中默认将所有方 Spring Cloud 达尔斯顿改变了赞成选择加入的方式。

Feign Hystrix 回退

Hystrix 支持回退的概念：当电路断开或出现错误时执行的默认代码路径。要为给定的`@FeignClient`启用回退，请将 `fallback` 属性设置为实现回退的类名。

```
@FeignClient(name = "hello", fallback =
HystrixClientFallback.class)
protected interface HystrixClient {
    @RequestMapping(method = RequestMethod.GET, value =
"/hello")
    Hello iFailSometimes();
}

static class HystrixClientFallback implements
HystrixClient {
    @Override
    public Hello iFailSometimes() {
        return new Hello("fallback");
    }
}
```

如果需要访问导致回退触发的原因，可以使用`@FeignClient` 内的 `fallbackFactory` 属性。

```
@FeignClient(name = "hello", fallbackFactory =
HystrixClientFallbackFactory.class)
protected interface HystrixClient {
        @RequestMapping(method = RequestMethod.GET, value =
"/hello")
        Hello iFailSometimes();
}

@Component
```

```
static class HystrixClientFallbackFactory implements
FallbackFactory<HystrixClient> {
	@Override
	public HystrixClient create(Throwable cause) {
		return new
HystrixClientWithFallBackFactory() {
			@Override
			public Hello iFailSometimes() {
				return new Hello("fallback;
reason was: " + cause.getMessage());
			}
		};
	}
}
```

警告	在 Feign 中执行回退以及 Hystrix 回退的工作方式存在局限性。当前返回 `com.netflix.rx.Observable` 的方法目前不支持回退。

Feign 和`@Primary`

当使用 Feign 与 Hystrix 回退时，在同一类型的 `ApplicationContext` 中有多个 bean。这将导致`@Autowired` 不起作用，因为没有一个 bean，或者标记为主。要解决这个问题，Spring Cloud Netflix 将所有 Feign 实例标记为 `@Primary`，所以 Spring Framework 将知道要注入哪个 bean。在某些情况下，这可能是不可取的。要关闭此行为，将`@FeignClient` 的 `primary` 属性设置为 false。

```
@FeignClient(name = "hello", primary = false)
public interface HelloClient {
	// methods here
}
```

Feign 继承支持

Feign 通过单继承接口支持样板 apis。这样就可以将常用操作分成方便的基本界面。

UserService.java

```
public interface UserService {

    @RequestMapping(method = RequestMethod.GET, value
="/users/{id}")
    User getUser(@PathVariable("id") long id);
}
```

UserResource.java

```
@RestController
public class UserResource implements UserService {

}
```

UserClient.java

```
package project.user;

@FeignClient("users")
public interface UserClient extends UserService {

}
```

注意	通常不建议在服务器和客户端之间共享接口。它引入了紧耦合，并且实际上并不适 映射不被继承）。

Feign 请求/响应压缩

您可以考虑为 Feign 请求启用请求或响应 GZIP 压缩。您可以通过启用其中一个属性来执行此操作：

```
feign.compression.request.enabled=true
feign.compression.response.enabled=true
```

Feign 请求压缩为您提供与您为 Web 服务器设置的设置相似的设置：

```
feign.compression.request.enabled=true
```

```
feign.compression.request.mime-
types=text/xml,application/xml,application/json
feign.compression.request.min-request-size=2048
```

这些属性可以让您对压缩介质类型和最小请求阈值长度有选择性。

Feign 日志记录

为每个创建的 Feign 客户端创建一个记录器。默认情况下，记录器的名称是用于创建 Feign 客户端的接口的完整类名。Feign 日志记录仅响应 `DEBUG` 级别。

application.yml

```
logging.level.project.user.UserClient: DEBUG
```

您可以为每个客户端配置的 `Logger.Level` 对象告诉 Feign 记录多少。选择是:

- `NONE`，无记录（**DEFAULT**）。
- `BASIC`，只记录请求方法和 URL 以及响应状态代码和执行时间。
- `HEADERS`，记录基本信息以及请求和响应标头。
- `FULL`，记录请求和响应的头文件，正文和元数据。

例如，以下将 `Logger.Level` 设置为 `FULL`:

```
@Configuration
public class FooConfiguration {
    @Bean
    Logger.Level feignLoggerLevel() {
        return Logger.Level.FULL;
    }
}
```

外部配置：Archaius

Netflix 客户端配置库 Archaius 它是所有 Netflix OSS 组件用于配置的库。

Archaius 是 Apache Commons Configuration 项目的扩展。它允许通过轮询源进行更改或将源更改推送到客户端来进行配置更新。Archaius 使用 Dynamic <Type> Property 类作为属性的句柄。

Archaius 示例

```
class ArchaiusTest {
    DynamicStringProperty myprop = DynamicPropertyFactory
            .getInstance()
            .getStringProperty("my.prop");

    void doSomething() {
        OtherClass.someMethod(myprop.get());
    }
}
```

Archaius 具有自己的一组配置文件和加载优先级。Spring 应用程序一般不应直接使用 Archaius，但本地仍然需要配置 Netflix 工具。Spring Cloud 具有 Spring 环境桥，所以 Archaius 可以从 Spring 环境读取属性。这允许 Spring Boot 项目使用正常的配置工具链，同时允许他们在文档中大部分配置 Netflix 工具。

路由器和过滤器：Zuul

路由在微服务体系结构的一个组成部分。例如，`/`可以映射到您的 Web 应用程序，`/api/users`映射到用户服务，并将`/api/shop`映射到商店服务。Zuul 是 Netflix 的基于 JVM 的路由器和服务器端负载均衡器。

Netflix 使用 Zuul 进行以下操作：

- 认证
- 洞察
- 压力测试
- 金丝雀测试
- 动态路由
- 服务迁移
- 负载脱落
- 安全
- 静态响应处理
- 主动/主动流量管理

Zuul 的规则引擎允许基本上写任何 JVM 语言编写规则和过滤器，内置 Java 和 Groovy。

注意	配置属性`zuul.max.host.connections`已被两个新属性`zuul.host.maxTota` `zuul.host.maxPerRouteConnections`替换，分别默认为 200 和 20。

注意 所有路由的默认 Hystrix 隔离模式（ExecutionIsolationStrategy）为 SEMAPHORE。`zuul.ribbonIsolationStrategy` 可以更改为 THREAD。

如何加入 Zuul

要在您的项目中包含 Zuul，请使用组 `org.springframework.cloud` 和 artifact id `spring-cloud-starter-zuul` 的启动器。有关 使用当前的 Spring Cloud 发布列表设置构建系统的详细信息，请参阅 Spring Cloud 项目页面。

嵌入式 Zuul 反向代理

Spring Cloud 已经创建了一个嵌入式 Zuul 代理，以简化 UI 应用程序想要代理对一个或多个后端服务的呼叫的非常常见的用例的开发。此功能对于用户界面对其所需的后端服务进行代理是有用的，避免了对所有后端独立管理 CORS 和验证问题的需求。

要启用它，使用`@EnableZuulProxy`注释 Spring Boot 主类，并将本地调用转发到相应的服务。按照惯例，具有 ID“用户”的服务将接收来自位于`/users`（具有前缀 stripped）的代理的请求。代理使用 Ribbon 来定位一个通过发现转发的实例，并且所有请求都以 hystrix 命令执行，所以故障将显示在 Hystrix 指标中，一旦电路打开，代理将不会尝试联系服务。

注意 Zuul 启动器不包括发现客户端，因此对于基于服务 ID 的路由，您还需要在类路径中

要跳过自动添加的服务，请将`zuul.ignored-services`设置为服务标识模式列表。如果一个服务匹配一个被忽略的模式，而且包含在明确配置的路由映射中，那么它将被无符号。例：

application.yml
```
zuul:
  ignoredServices: '*'
  routes:
    users: /myusers/**
```

在此示例中，**除**“用户”**之外**，所有服务都被忽略。

要扩充或更改代理路由，可以添加如下所示的外部配置：

application.yml
```
zuul:
  routes:
    users: /myusers/**
```

这意味着对“/ myusers”的 http 呼叫转发到“用户”服务（例如“/ myusers / 101”转发到“/ 101”）。

要获得对路由的更细粒度的控制，您可以独立地指定路径和 serviceId：

application.yml
```
zuul:
  routes:
    users:
      path: /myusers/**
      serviceId: users_service
```

这意味着对“/ myusers”的 http 呼叫转发到“users_service”服务。路由必须有一个“路径”，可以指定为蚂蚁样式模式，所以“/ myusers / *”只匹配一个级别，但“/ myusers / **”分层匹配。

后端的位置可以被指定为“serviceId”（用于发现的服务）或“url”（对于物理位置），例如

application.yml
```
zuul:
  routes:
    users:
      path: /myusers/**
      url: http://example.com/users_service
```

这些简单的 URL 路由不会被执行为 `HystrixCommand`，也不能使用 Ribbon 对多个 URL 进行负载平衡。为此，请指定 service-route 并为 serviceId 配置 Ribbon 客户端（目前需要在 Ribbon 中禁用 Eureka 支持：详见上文），例如

application.yml
```
zuul:
  routes:
    users:
      path: /myusers/**
      serviceId: users

ribbon:
  eureka:
    enabled: false

users:
  ribbon:
    listOfServers: example.com,google.com
```

您可以使用 regexmapper 在 serviceId 和路由之间提供约定。它使用名为 group 的正则表达式从 serviceId 中提取变量并将它们注入到路由模式中。

ApplicationConfiguration.java
```
@Bean
public PatternServiceRouteMapper serviceRouteMapper() {
    return new PatternServiceRouteMapper(
        "(?<name>^.+)-(?<version>v.+$)",
        "${version}/${name}");
```

```
}
```

这意味着 serviceId“myusers-v1”将被映射到路由“/ v1 / myusers / **”。任何正则表达式都被接受，但所有命名组都必须存在于 servicePattern 和 routePattern 中。如果 servicePattern 与 serviceId 不匹配，则使用默认行为。在上面的示例中，serviceId“myusers”将被映射到路由“/ myusers / **”（检测不到版本）此功能默认禁用，仅适用于已发现的服务。

要为所有映射添加前缀，请将 `zuul.prefix` 设置为一个值，例如`/api`。默认情况下，请求被转发之前，代理前缀被删除（使用 `zuul.stripPrefix=false` 关闭此行为）。您还可以关闭从各路线剥离服务特定的前缀，例如

application.yml

```
zuul:
 routes:
   users:
     path: /myusers/**
     stripPrefix: false
```

注意	`zuul.stripPrefix` 仅适用于 `zuul.prefix` 中设置的前缀。它对给定路由 `path`

在本示例中，对“/ myusers / 101”的请求将转发到“/ myusers / 101”上的“users”服务。

`zuul.routes` 条目实际上绑定到类型为 `ZuulProperties` 的对象。如果您查看该对象的属性，您将看到它还具有“可重试”标志。将该标志设置为“true”使 Ribbon 客户端自动重试失败的请求（如果需要，可以使用 Ribbon 客户端配置修改重试操作的参数）。

默认情况下，将`X-Forwarded-Host`标头添加到转发的请求中。关闭 set`zuul.addProxyHeaders = false`。默认情况下，前缀路径被删除，对后端的请求会拾取一个标题“X-Forwarded-Prefix”（上述示例中的“/ myusers”）。

如果您设置默认路由（“/”），则`@EnableZuulProxy`的应用程序可以作为独立服务器，例如`zuul.route.home: /`将路由所有流量（即“/ **”）到“home”服务。

如果需要更细粒度的忽略，可以指定要忽略的特定模式。在路由位置处理开始时评估这些模式，这意味着前缀应包含在模式中以保证匹配。忽略的模式跨越所有服务，并取代任何其他路由规范。

application.yml

```
zuul:
 ignoredPatterns: /**/admin/**
 routes:
   users: /myusers/**
```

这意味着诸如“/ myusers / 101”的所有呼叫将被转发到“用户”服务上的“/ 101”。但是包含“/ admin /”的呼叫将无法解决。

警告	如果您需要您的路由保留订单，则需要使用 YAML 文件，因为使用属性文件将会丢失

application.yml

```
zuul:
 routes:
   users:
     path: /myusers/**
   legacy:
     path: /**
```

如果要使用属性文件，则 `legacy` 路径可能会在 `users` 路径前面展开，从而使 `users` 路径不可达。

Zuul Http 客户端

zuul 使用的默认 HTTP 客户端现在由 Apache HTTP Client 支持，而不是不推荐使用的 Ribbon `RestClient`。要分别使用 `RestClient` 或使用 `okhttp3.OkHttpClient` 集合 `ribbon.restclient.enabled=true` 或 `ribbon.okhttp.enabled=true`。

Cookie 和敏感标题

在同一个系统中的服务之间共享标题是可行的，但是您可能不希望敏感标头泄漏到外部服务器的下游。您可以在路由配置中指定被忽略头文件列表。Cookies 起着特殊的作用，因为它们在浏览器中具有明确的语义，并且它们总是被视为敏感的。如果代理的消费者是浏览器，则下游服务的 cookie 也会导致用户出现问题，因为它们都被混淆（所有下游服务看起来都是来自同一个地方）。

如果您对服务的设计非常谨慎，例如，如果只有一个下游服务设置了 Cookie，那么您可能可以让他们从后台一直到调用者。另外，如果您的代理设置 cookie 和所有后台服务都是同一系统的一部分，那么简单地共享它们就可以自然（例如使用 Spring Session 将它们链接到一些共享状态）。除此之外，由下游服务设置的任何 Cookie 可能对呼叫者来说都不是很有用，因此建议您将（至少）“Set-

Cookie”和“Cookie”设置为不属于您的域名。即使**是**属于您域名的路线，请尝试仔细考虑允许 Cookie 在代理之间流动的含义。

灵敏头可以配置为每个路由的逗号分隔列表，例如

application.yml

```
zuul:
 routes:
   users:
     path: /myusers/**
     sensitiveHeaders: Cookie,Set-Cookie,Authorization
     url: https://downstream
```

注意	这是 `sensitiveHeaders` 的默认值，因此您不需要设置它，除非您希望它不同。注 的新功能（1.0 中，用户无法控制标题，所有 Cookie 都在两个方向上流动）。

`sensitiveHeaders` 是一个黑名单，默认值不为空，所以要使 Zuul 发送所有标题（“被忽略”除外），您必须将其显式设置为空列表。如果您要将 Cookie 或授权标头传递到后端，这是必要的。例:

application.yml

```
zuul:
 routes:
   users:
     path: /myusers/**
     sensitiveHeaders:
     url: https://downstream
```

也可以通过设置 `zuul.sensitiveHeaders` 来全局设置敏感标题。如果在路由上设置 `sensitiveHeaders`，则将覆盖全局 `sensitiveHeaders` 设置。

被忽略的标题

除了每个路由的敏感标头，您还可以为与下游服务交互期间应该丢弃的值（请求和响应）设置全局值为 `zuul.ignoredHeaders`。默认情况下，如果 Spring 安全性不在类路径上，则它们是空的，否则它们被初始化为由 Spring Security 指定的一组众所周知的“安全性”头（例如涉及缓存）。在这种情况下的假设是下游服务可能也添加这些头，我们希望代理的值。为了不丢弃这些众所周知的安全标头，只要 Spring 安全性在类路径上，您可以将 `zuul.ignoreSecurityHeaders` 设置为 `false`。如果您禁用 Spring 安全性中的 HTTP 安全性响应头，并希望由下游服务提供的值，这可能很有用

路线端点

如果您在 Spring Boot 执行器中使用`@EnableZuulProxy`，您将启用（默认情况下）另一个端点，通过 HTTP 可用`/routes`。到此端点的 GET 将返回映射路由的列表。POST 将强制刷新现有路由（例如，如果服务目录中有更改）。您可以通过将 `endpoints.routes.enabled` 设置为 `false` 来禁用此端点。

注意	路由应自动响应服务目录中的更改，但 POST 到/路由是强制更改立即发生的一种方

扼杀模式和本地前进

迁移现有应用程序或 API 时的常见模式是“扼杀”旧端点，用不同的实现慢慢替换它们。Zuul 代理是一个有用的工具，因为您可以使用它来处理来自旧端点的客户端的所有流量，但将一些请求重定向到新端点。

示例配置:

application.yml
```
zuul:
  routes:
    first:
      path: /first/**
      url: http://first.example.com
    second:
      path: /second/**
      url: forward:/second
    third:
      path: /third/**
      url: forward:/3rd
    legacy:
      path: /**
      url: http://legacy.example.com
```

在这个例子中，我们扼杀了“遗留”应用程序，该应用程序映射到所有与其他模式不匹配的请求。`/first/**`中的路径已被提取到具有外部 URL 的新服务中。并且`/second/**`中的路径被转发，以便它们可以在本地处理，例如具有正常的 Spring `@RequestMapping`。`/third/**`中的路径也被转发，但具有不同的前缀（即`/third/foo` 转发到`/3rd/foo`）。

注意	被忽略的模式并不完全被忽略，它们只是不被代理处理（因此它们也被有效地转发

通过 Zuul 上传文件

如果您`@EnableZuulProxy` 您可以使用代理路径上传文件，只要文件很小，它就应该工作。对于大文件，有一个替代路径绕过“/ zuul / *”中的 Spring `DispatcherServlet`（以避免多部分处理）。也就是说，如果`zuul.routes.customers=/customers/**`则可以将大文件发送到“/ zuul /

customers / *”。servlet 路径通过 `zuul.servletPath` 进行外部化。如果代理路由引导您通过 Ribbon 负载均衡器，例如，超大文件也将需要提升超时设置

application.yml

```
hystrix.command.default.execution.isolation.thread.timeou
tInMilliseconds: 60000
ribbon:
  ConnectTimeout: 3000
  ReadTimeout: 60000
```

请注意，要使用大型文件进行流式传输，您需要在请求中使用分块编码（某些浏览器默认情况下不会执行）。例如在命令行：

```
$ curl -v -H "Transfer-Encoding: chunked" \
    -F "file=@mylarge.iso" localhost:9999/zuul/simple/file
```

查询字符串编码

处理传入的请求时，查询参数被解码，因此可以在 Zuul 过滤器中进行修改。然后在路由过滤器中构建后端请求时重新编码它们。如果使用 Javascript 的 `encodeURIComponent()` 方法编码，结果可能与原始输入不同。虽然这在大多数情况下不会出现任何问题，但一些 Web 服务器可以用复杂查询字符串的编码来挑选。

要强制查询字符串的原始编码，可以将特殊标志传递给 `ZuulProperties`，以便查询字符串与 `HttpServletRequest::getQueryString` 方法相同：

application.yml

```
 zuul:
  forceOriginalQueryStringEncoding: true
```

注意：此特殊标志仅适用于 `SimpleHostRoutingFilter`，您可以使用 `RequestContext.getCurrentContext().setRequestQueryParams(someOverriddenParameters)`轻松覆盖查询参数，因为查询字符串现在直接在原始的 `HttpServletRequest` 上获取。

普通嵌入 Zuul

如果您使用`@EnableZuulServer`（而不是`@EnableZuulProxy`），您也可以运行不带代理的 Zuul 服务器，或者有选择地切换代理平台的部分。您添加到 `ZuulFilter` 类型的应用程序的任何 bean 都将自动安装，与 `@EnableZuulProxy`一样，但不会自动添加任何代理过滤器。

在这种情况下，仍然通过配置“zuul.routes。*”来指定进入 Zuul 服务器的路由，但没有服务发现和代理，所以“serviceId”和“url”设置将被忽略。例如：

application.yml
```
zuul:
 routes:
   api: /api/**
```

将“/ api / **”中的所有路径映射到 Zuul 过滤器链。

禁用 Zuul 过滤器

Spring Cloud 的 Zuul 在代理和服务器模式下默认启用了多个 `ZuulFilter` bean。有关启用的可能过滤器，请参阅 zuul 过滤器包。如果要禁用它，只需设置

`zuul.<SimpleClassName>.<filterType>.disable=true`。按照惯例，`filters` 之后的包是 Zuul 过滤器类型。例如，禁用 `org.springframework.cloud.netflix.zuul.filters.post.SendResponseFilter` 设置 `zuul.SendResponseFilter.post.disable=true`。

为路线提供 Hystrix 回退

当 Zuul 中给定路由的电路跳闸时，您可以通过创建类型为 `ZuulFallbackProvider` 的 bean 来提供回退响应。在这个 bean 中，您需要指定回退的路由 ID，并提供返回的 `ClientHttpResponse` 作为后备。这是一个非常简单的 `ZuulFallbackProvider` 实现。

```
class MyFallbackProvider implements ZuulFallbackProvider
{
    @Override
    public String getRoute() {
        return "customers";
    }

    @Override
    public ClientHttpResponse fallbackResponse() {
        return new ClientHttpResponse() {
            @Override
            public HttpStatus getStatusCode() throws
IOException {
                return HttpStatus.OK;
            }

            @Override
            public int getRawStatusCode() throws
IOException {
                return 200;
            }

            @Override
```

```
            public String getStatusText() throws
IOException {
                return "OK";
            }

            @Override
            public void close() {

            }

            @Override
            public InputStream getBody() throws IOException
{
                return new
ByteArrayInputStream("fallback".getBytes());
            }

            @Override
            public HttpHeaders getHeaders() {
                HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION_JSON);
                return headers;
            }
        };
    }
}
```

这里是路由配置的样子。

```
zuul:
  routes:
    customers: /customers/**
```

如果您希望为所有路由提供默认的回退，您可以创建一个类型为

`ZuulFallbackProvider`的 bean，并且`getRoute`方法返回`*`或`null`。

```
class MyFallbackProvider implements ZuulFallbackProvider
{
    @Override
    public String getRoute() {
        return "*";
```

```
    }

    @Override
    public ClientHttpResponse fallbackResponse() {
        return new ClientHttpResponse() {
            @Override
            public HttpStatus getStatusCode() throws
IOException {
                return HttpStatus.OK;
            }

            @Override
            public int getRawStatusCode() throws
IOException {
                return 200;
            }

            @Override
            public String getStatusText() throws
IOException {
                return "OK";
            }

            @Override
            public void close() {

            }

            @Override
            public InputStream getBody() throws IOException
{
                return new
ByteArrayInputStream("fallback".getBytes());
            }

            @Override
            public HttpHeaders getHeaders() {
                HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION_JSON);
                return headers;
            }
        };
    }
```

```
}
```

Zuul 开发人员指南

有关 Zuul 如何工作的一般概述，请参阅 Zuul 维基。

Zuul Servlet

Zuul 被实现为 Servlet。对于一般情况，Zuul 嵌入到 Spring 调度机制中。这允许 Spring MVC 控制路由。在这种情况下，Zuul 被配置为缓冲请求。如果需要通过 Zuul 不缓冲请求（例如大文件上传），Servlet 也将安装在 Spring 调度程序之外。默认情况下，它位于`/zuul`。可以使用 `zuul.servlet-path` 属性更改此路径。

Zuul RequestContext

要在过滤器之间传递信息，Zuul 使用 a `RequestContext`。其数据按照每个请求的 `ThreadLocal` 进行。关于路由请求，错误以及实际 `HttpServletRequest` 和 `HttpServletResponse` 的路由信息。`RequestContext` 扩展 `ConcurrentHashMap`，所以任何东西都可以存储在上下文中。`FilterConstants` 包含由 Spring Cloud Netflix 安装的过滤器使用的密钥（稍后再安装）。

`@EnableZuulProxy` 与 `@EnableZuulServer`

Spring Cloud Netflix 根据使用何种注释来启用 Zuul 安装多个过滤器。`@EnableZuulProxy` 是 `@EnableZuulServer` 的超集。换句话说，

`@EnableZuulProxy`包含`@EnableZuulServer`安装的所有过滤器。“代理”中的其他过滤器启用路由功能。如果你想要一个“空白”Zuul，你应该使用`@EnableZuulServer`。

`@EnableZuulServer`过滤器

创建从 Spring Boot 配置文件加载路由定义的`SimpleRouteLocator`。

安装了以下过滤器（正常 Spring 豆类）：

前置过滤器

- `ServletDetectionFilter`：检测请求是否通过 Spring 调度程序。使用键`FilterConstants.IS_DISPATCHER_SERVLET_REQUEST_KEY`设置布尔值。
- `FormBodyWrapperFilter`：解析表单数据，并对下游请求进行重新编码。
- `DebugFilter`：如果设置`debug`请求参数，则此过滤器将`RequestContext.setDebugRouting()`和`RequestContext.setDebugRequest()`设置为 true。

路由过滤器

- `SendForwardFilter`：此过滤器使用 Servlet `RequestDispatcher`转发请求。转发位置存储在`RequestContext`属性

`FilterConstants.FORWARD_TO_KEY`中。这对于转发到当前应用程序中的端点很有用。

过滤器:

- `SendResponseFilter`：将代理请求的响应写入当前响应。

错误过滤器:

- `SendErrorFilter`：如果`RequestContext.getThrowable()`不为null，则转发到/错误（默认情况下）。可以通过设置`error.path`属性来更改默认转发路径（`/error`）。

`@EnableZuulProxy`过滤器

创建从`DiscoveryClient`（如 Eureka）以及属性加载路由定义的`DiscoveryClientRouteLocator`。每个`serviceId`从`DiscoveryClient`创建路由。随着新服务的添加，路由将被刷新。

除了上述过滤器之外，还安装了以下过滤器（正常 Spring 豆类）：

前置过滤器

- `PreDecorationFilter`：此过滤器根据提供的`RouteLocator`确定在哪里和如何路由。它还为下游请求设置各种与代理相关的头。

路由过滤器

- `RibbonRoutingFilter`：此过滤器使用 Ribbon，Hystrix 和可插拔 HTTP 客户端发送请求。服务 ID 位于 `RequestContext` 属性 `FilterConstants.SERVICE_ID_KEY` 中。此过滤器可以使用不同的 HTTP 客户端。他们是：
  - Apache `HttpClient`。这是默认的客户端。
  - Squareup `OkHttpClient` v3。通过在类路径上设置 `com.squareup.okhttp3:okhttp` 库并设置 `ribbon.okhttp.enabled=true` 来启用此功能。
  - Netflix Ribbon HTTP 客户端。这可以通过设置 `ribbon.restclient.enabled=true` 来启用。这个客户端有限制，比如它不支持 PATCH 方法，还有内置的重试。
- `SimpleHostRoutingFilter`：此过滤器通过 Apache HttpClient 发送请求到预定的 URL。URL 位于 `RequestContext.getRouteHost()`。

自定义 Zuul 过滤示例

以下大部分以下“如何撰写”示例都包含示例 Zuul 过滤器项目。还有一些操作该存储库中的请求或响应正文的例子。

如何编写预过滤器

前置过滤器用于设置 `RequestContext` 中的数据，用于下游的过滤器。主要用例是设置路由过滤器所需的信息。

```
public class QueryParamPreFilter extends ZuulFilter {
```

```
        @Override
        public int filterOrder() {
                return PRE_DECORATION_FILTER_ORDER - 1; //
run before PreDecoration
        }

        @Override
        public String filterType() {
                return PRE_TYPE;
        }

        @Override
        public boolean shouldFilter() {
                RequestContext ctx =
RequestContext.getCurrentContext();
                return !ctx.containsKey(FORWARD_TO_KEY) // a
filter has already forwarded

        && !ctx.containsKey(SERVICE_ID_KEY); // a filter
has already determined serviceId
        }
    @Override
    public Object run() {
        RequestContext ctx =
RequestContext.getCurrentContext();
                HttpServletRequest request =
ctx.getRequest();
                if (request.getParameter("foo") != null) {
                    // put the serviceId in `RequestContext`
                ctx.put(SERVICE_ID_KEY,
request.getParameter("foo"));
        }
        return null;
    }
}
```

上面的过滤器从`foo`请求参数填充`SERVICE_ID_KEY`。实际上，做这种直接映射并不是一个好主意，而是从`foo`的值来查看服务 ID。

现在填写`SERVICE_ID_KEY`，`PreDecorationFilter`将不会运行，`RibbonRoutingFilter`将会。如果您想要路由到完整的网址，请改用`ctx.setRouteHost(url)`。

要修改路由过滤器将转发的路径，请设置`REQUEST_URI_KEY`。

如何编写路由过滤器

路由过滤器在预过滤器之后运行，并用于向其他服务发出请求。这里的大部分工作是将请求和响应数据转换到客户端所需的模型。

```
public class OkHttpRoutingFilter extends ZuulFilter {
	@Autowired
	private ProxyRequestHelper helper;

	@Override
	public String filterType() {
		return ROUTE_TYPE;
	}

	@Override
	public int filterOrder() {
		return SIMPLE_HOST_ROUTING_FILTER_ORDER - 1;
	}

	@Override
	public boolean shouldFilter() {
		return
RequestContext.getCurrentContext().getRouteHost() != null
				&&
RequestContext.getCurrentContext().sendZuulResponse();
	}

    @Override
    public Object run() {
		OkHttpClient httpClient = new
OkHttpClient.Builder()
				// customize
```

```
                        .build();

            RequestContext context =
RequestContext.getCurrentContext();
            HttpServletRequest request =
context.getRequest();

            String method = request.getMethod();

            String uri =
this.helper.buildZuulRequestURI(request);

            Headers.Builder headers = new
Headers.Builder();
            Enumeration<String> headerNames =
request.getHeaderNames();
            while (headerNames.hasMoreElements()) {
                  String name =
headerNames.nextElement();
                  Enumeration<String> values =
request.getHeaders(name);

                  while (values.hasMoreElements()) {
                        String value =
values.nextElement();
                        headers.add(name, value);
                  }
            }

            InputStream inputStream =
request.getInputStream();

            RequestBody requestBody = null;
            if (inputStream != null &&
HttpMethod.permitsRequestBody(method)) {
                  MediaType mediaType = null;
                  if (headers.get("Content-Type") !=
null) {
                        mediaType =
MediaType.parse(headers.get("Content-Type"));
                  }
                  requestBody =
RequestBody.create(mediaType,
StreamUtils.copyToByteArray(inputStream));
```

```
            }

            Request.Builder builder = new
Request.Builder()
                         .headers(headers.build())
                         .url(uri)
                         .method(method, requestBody);

            Response response =
httpClient.newCall(builder.build()).execute();

            LinkedMultiValueMap<String, String>
responseHeaders = new LinkedMultiValueMap<>();

            for (Map.Entry<String, List<String>> entry :
response.headers().toMultimap().entrySet()) {
                  responseHeaders.put(entry.getKey(),
entry.getValue());
            }

            this.helper.setResponse(response.code(),
response.body().byteStream(),
                        responseHeaders);
            context.setRouteHost(null); // prevent
SimpleHostRoutingFilter from running
            return null;
    }
}
```

上述过滤器将 Servlet 请求信息转换为 OkHttp3 请求信息，执行 HTTP 请求，然后将 OkHttp3 响应信息转换为 Servlet 响应。警告：此过滤器可能有错误，但功能不正确。

如何编写过滤器

后置过滤器通常操纵响应。在下面的过滤器中，我们添加一个随机 `UUID` 作为 `X-Foo` 头。其他操作，如转换响应体，要复杂得多，计算密集。

```
public class AddResponseHeaderFilter extends ZuulFilter {
```

```
	@Override
	public String filterType() {
		return POST_TYPE;
	}

	@Override
	public int filterOrder() {
		return SEND_RESPONSE_FILTER_ORDER - 1;
	}

	@Override
	public boolean shouldFilter() {
		return true;
	}

	@Override
	public Object run() {
		RequestContext context =
RequestContext.getCurrentContext();
	HttpServletResponse servletResponse =
context.getResponse();
		servletResponse.addHeader("X-Foo",
UUID.randomUUID().toString());
		return null;
	}
}
```

Zuul 错误如何工作

如果在 Zuul 过滤器生命周期的任何部分抛出异常，则会执行错误过滤器。

`SendErrorFilter` 只有 `RequestContext.getThrowable()` 不是 `null` 才会运行。然后在请求中设置特定的 `javax.servlet.error.*` 属性，并将请求转发到 Spring Boot 错误页面。

Zuul 渴望应用程序上下文加载

Zuul 内部使用 Ribbon 调用远程 URL，并且 Ribbon 客户端默认在第一次调用时由 Spring Cloud 加载。可以使用以下配置更改 Zuul 的此行为，并将导致在应用程序启动时，子 Ribbon 相关的应用程序上下文正在加载。

application.yml

```
zuul:
  ribbon:
    eager-load:
      enabled: true
```

Polyglot 支持 Sidecar

你有没有非 jvm 的语言你想利用 Eureka，Ribbon 和配置服务器？Netflix Prana 启发了 Spring Cloud Netflix Sidecar 。它包含一个简单的 http api 来获取给定服务的所有实例（即主机和端口）。您还可以通过从 Eureka 获取其路由条目的嵌入式 Zuul 代理来代理服务调用。可以通过主机查找或通过 Zuul 代理访问 Spring Cloud Config 服务器。非 jvm 应用程序应该执行健康检查，以便 Sidecar 可以向应用程序启动或关闭时向 eureka 报告。

要在项目中包含 Sidecar，使用组 `org.springframework.cloud` 和 artifact id `spring-cloud-netflix-sidecar` 的依赖关系。

要启用 Sidecar，请使用`@EnableSidecar` 创建 Spring Boot 应用程序。此注释包括`@EnableCircuitBreaker`，`@EnableDiscoveryClient` 和 `@EnableZuulProxy`。在与非 jvm 应用程序相同的主机上运行生成的应用程序。

要配置侧车将`sidecar.port`和`sidecar.health-uri`添加到`application.yml`。`sidecar.port`属性是非 jvm 应用程序正在侦听的端口。这样，Sidecar 可以使用 Eureka 正确注册该应用。`sidecar.health-uri`是可以在非 jvm 应用程序上访问的，可以模拟 Spring Boot 健康指标。它应该返回一个 json 文档，如下所示：

健康-URI 文档

```
{
  "status":"UP"
}
```

以下是 Sidecar 应用程序的 application.yml 示例：

application.yml

```
server:
  port: 5678
spring:
  application:
    name: sidecar

sidecar:
  port: 8000
  health-uri: http://localhost:8000/health.json
```

`DiscoveryClient.getInstances()`方法的 api 为`/hosts/{serviceId}`。以下是`/hosts/customers`在不同主机上返回两个实例的示例响应。这个 api 可以访问http://localhost:5678/hosts/{serviceId}的非 jvm 应用程序（如果侧面在端口 5678 上）。

/主机/客户

```
[
    {
        "host": "myhost",
```

```
        "port": 9000,
        "uri": "http://myhost:9000",
        "serviceId": "CUSTOMERS",
        "secure": false
    },
    {
        "host": "myhost2",
        "port": 9000,
        "uri": "http://myhost2:9000",
        "serviceId": "CUSTOMERS",
        "secure": false
    }
]
```

Zuul 代理自动将 eureka 中已知的每个服务的路由添加到/<serviceId>，因此客户服务可在/customers 获得。非 jvm 应用程序可以通过 http://localhost:5678/customers 访问客户服务（假设边界正在侦听端口 5678）。

如果配置服务器已注册到 Eureka，则非 jvm 应用程序可以通过 Zuul 代理访问它。如果 ConfigServer 的 serviceId 为 configserver 且端口 5678 为 Sidecar，则可以在 http：// localhost：5678 / configserver

非 jvm 应用程序可以利用 Config Server 返回 YAML 文档的功能。例如，调用 http://sidecar.local.spring.io:5678/configserver/default-master.yml 可能会导致一个 YAML 文档，如下所示

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
  password: password
info:
  description: Spring Cloud Samples
  url: https://github.com/spring-cloud-samples
```

RxJava 与 Spring MVC

Spring Cloud Netflix 包括 RxJava。

RxJava 是 Reactive Extensions 的 Java VM 实现：用于通过使用 observable 序列来构建异步和基于事件的程序的库。

Spring Cloud Netflix 支持从 Spring MVC 控制器返回 `rx.Single` 对象。它还支持对服务器发送事件（SSE）使用 `rx.Observable` 对象。如果您的内部 API 已经使用 RxJava 构建（参见 Feign Hystrix 支持示例），这可能非常方便。

以下是使用 `rx.Single` 的一些示例：

```
@RequestMapping(method = RequestMethod.GET, value =
"/single")
public Single<String> single() {
	return Single.just("single value");
}

@RequestMapping(method = RequestMethod.GET, value =
"/singleWithResponse")
public ResponseEntity<Single<String>>
singleWithResponse() {
	return new ResponseEntity<>(Single.just("single
value"),
			HttpStatus.NOT_FOUND);
}

@RequestMapping(method = RequestMethod.GET, value =
"/singleCreatedWithResponse")
public Single<ResponseEntity<String>>
singleOuterWithResponse() {
	return Single.just(new ResponseEntity<>("single
value", HttpStatus.CREATED));
}
```

```
@RequestMapping(method = RequestMethod.GET, value =
"/throw")
public Single<Object> error() {
       return Single.error(new
RuntimeException("Unexpected"));
}
```

如果您有`Observable`而不是单个，则可以使用`.toSingle()`

或`.toList().toSingle()`。这里有些例子：

```
@RequestMapping(method = RequestMethod.GET, value =
"/single")
public Single<String> single() {
    return Observable.just("single value").toSingle();
}

@RequestMapping(method = RequestMethod.GET, value =
"/multiple")
public Single<List<String>> multiple() {
    return Observable.just("multiple",
"values").toList().toSingle();
}

@RequestMapping(method = RequestMethod.GET, value =
"/responseWithObservable")
public ResponseEntity<Single<String>>
responseWithObservable() {

    Observable<String> observable =
Observable.just("single value");
    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(APPLICATION_JSON_UTF8);
    return new ResponseEntity<>(observable.toSingle(),
headers, HttpStatus.CREATED);
}

@RequestMapping(method = RequestMethod.GET, value =
"/timeout")
public Observable<String> timeout() {
    return Observable.timer(1, TimeUnit.MINUTES).map(new
Func1<Long, String>() {
        @Override
```

```
        public String call(Long aLong) {
            return "single value";
        }
    });
}
```

如果您有流式端点和客户端，SSE 可以是一个选项。要将 `rx.Observable` 转换为 Spring `SseEmitter` 使用 `RxResponse.sse()`。这里有些例子:

```
@RequestMapping(method = RequestMethod.GET, value =
"/sse")
public SseEmitter single() {
        return RxResponse.sse(Observable.just("single
value"));
}

@RequestMapping(method = RequestMethod.GET, value =
"/messages")
public SseEmitter messages() {
        return RxResponse.sse(Observable.just("message 1",
"message 2", "message 3"));
}

@RequestMapping(method = RequestMethod.GET, value =
"/events")
public SseEmitter event() {
        return RxResponse.sse(APPLICATION_JSON_UTF8,
                        Observable.just(new EventDto("Spring
io", getDate(2016, 5, 19)),
                                        new
EventDto("SpringOnePlatform", getDate(2016, 8, 1))));
}
```

指标：Spectator，Servo 和 Atlas

当一起使用时，Spectator / Servo 和 Atlas 提供了近乎实时的操作洞察平台。

Netflix 的度量收集库 Spectator 和 Servo Atlas 是用于管理维度时间序列数据的 Netflix 指标后端。

Servo 为 Netflix 服务了好几年，仍然可以使用，但逐渐被淘汰出局 Spectator，这只适用于 Java 8. Spring Cloud Netflix 提供了支持，但 Java 8 鼓励基于应用的应用程序使用 Spectator。

维度与层次度量

Spring Boot 执行器指标是层次结构，指标只能由名称分隔。这些名称通常遵循将密钥/值属性对（维）嵌入到以句点分隔的名称中的命名约定。考虑以下两个端点（root 和 star-star）的指标：

```
{
    "counter.status.200.root": 20,
    "counter.status.400.root": 3,
    "counter.status.200.star-star": 5,
}
```

第一个指标给出了每单位时间内针对根端点的成功请求的归一化计数。但是如果系统有 20 个端点，并且想要获得针对所有端点的成功请求计数呢？一些分级度量后端将允许您指定一个通配符，例如 `counter.status.200.`**，它将读取所有 20 个指标并聚合结果。或者，您可以提供** `HandlerInterceptorAdapter` **拦截并记录所有成功请求的** `counter.status.200.all` **等指标，而不考虑端点，但现在您必须编写 20 + 1 个不同的指标。同样，如果您想知道服务中所有端点的成功请求总数，您可以指定一个通配符，例如** `counter.status.2` `.*`。

即使在分级度量后端的通配符支持的情况下，命名一致性也是困难的。具体来说，这些标签在名称字符串中的位置可能会随着时间而滑落，从而导致查询错 例如，假设我们为上述 HTTP 方法添加了一个额外的维度。那么 `counter.status.200.root` 成为 `counter.status.200.method.get.root` 等等。我们的 `counter.status.200.*` 突然不再具有相同的语义。此外，如果新的维度在整个代码库中不均匀地应用，某些查询可能会变得不可能。这可以很快失控。

Netflix 指标被标记（又称维度）。每个指标都有一个名称，但是这个单一的命名度量可以包含多个统计信息和"标签"键/值对，这允许更多的查询灵活性。实际上统计本身就是记录在一个特殊的标签上。

使用 Netflix Servo 或 Spectator 记录，上述根端点的计时器包含每个状态码的 4 个统计信息，其中计数统计信息与 Spring Boot 执行器计数器相同。如果到目前为止，我们遇到了 HTTP 200 和 400，将有 8 个可用数据点：

```
{
    "root(status=200,stastic=count)": 20,
    "root(status=200,stastic=max)": 0.7265630630000001,
    "root(status=200,stastic=totalOfSquares)":
0.04759702862580789,
    "root(status=200,stastic=totalTime)":
0.2093076914666667,
    "root(status=400,stastic=count)": 1,
    "root(status=400,stastic=max)": 0,
    "root(status=400,stastic=totalOfSquares)": 0,
    "root(status=400,stastic=totalTime)": 0,
}
```

默认度量集合

没有任何附加依赖或配置，基于 Spring Cloud 的服务将自动配置 Servo `MonitorRegistry`，并开始收集每个 Spring MVC 请求的指标。默认情况下，将为每个 MVC 请求记录名称为 `rest` 的 Servo 定时器，其标记为：

1. HTTP 方法
2. HTTP 状态（例如 200,400,500）
3. URI（如果 URI 为空，则为“root”），为 Atlas
4. 异常类名称，如果请求处理程序抛出异常
5. 如果在请求上设置了匹配 `netflix.metrics.rest.callerHeader` 的密钥的请求头，则呼叫者。`netflix.metrics.rest.callerHeader` 没有默认键。如果您希望收集来电者信息，则必须将其添加到应用程序属性中。

设置 `netflix.metrics.rest.metricName` 属性将度量值的名称从 `rest` 更改为您提供的名称。

如果 Spring AOP 已启用，并且 `org.aspectj:aspectjweaver` 存在于您的运行时类路径上，则 Spring Cloud 还将收集每个使用 `RestTemplate` 进行的客户端调用的指标。将为每个具有以下标签的 MVC 请求记录名称为 `restclient` 的 Servo 定时器：

1. HTTP 方法

2. HTTP 状态（例如 200,400,500），如果响应返回为空，则为 “CLIENT_ERROR”;如果在执行 `RestTemplate` 方法期间发生 `IOException`，则为“IO_ERROR”
3. URI，为 Atlas
4. 客户名称

警告	避免在 `RestTemplate` 内使用硬编码的 url 参数。定位动态端点时使用 URL 变量。个网址视为唯一密钥的潜在 “GC 覆盖限制达到” 问题。

```
// recommended
String orderid = "1";
restTemplate.getForObject("http://testeurekabrixtonclient
/orders/{orderid}", String.class, orderid)

// avoid
restTemplate.getForObject("http://testeurekabrixtonclient
/orders/1", String.class)
```

指标集：Spectator

要启用 Spectator 指标，请在 `spring-boot-starter-spectator` 上包含依赖关系：

```
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-
spectator</artifactId>
    </dependency>
```

在 Spectator 说明中，仪表是一个命名，打字和标记的配置，而指标表示给定仪表在某个时间点的值。Spectator 米由注册表创建和控制，注册表目前有几个不同的实现。Spectator 提供 4 米类型：计数器，定时器，量规和分配摘要。

Spring Cloud Spectator 集成为您配置可注入的 `com.netflix.spectator.api.Registry` 实例。具体来说，它配置一个 `ServoRegistry` 实例，以统一 REST 度量标准的集合，并将度量标准导出到 Servo API 下的 Atlas 后端。实际上，这意味着您的代码可能会使用 Servo 显示器和 Spectator 米的混合，并且都将由 Spring Boot Actuator `MetricReader` 实例舀取，并将两者都发送到 Atlas 后端

Spectator 柜台

计数器用于测量某些事件发生的速率。

```
// create a counter with a name and a set of tags
Counter counter = registry.counter("counterName",
"tagKey1", "tagValue1", ...);
counter.increment(); // increment when an event occurs
counter.increment(10); // increment by a discrete amount
```

计数器记录单个时间归一化统计量。

Spectator 计时器

一个计时器用于测量一些事件需要多长时间。Spring Cloud 自动记录 Spring MVC 请求和有条件 `RestTemplate` 请求的定时器，稍后可用于为请求相关指标创建仪表板，如延迟：

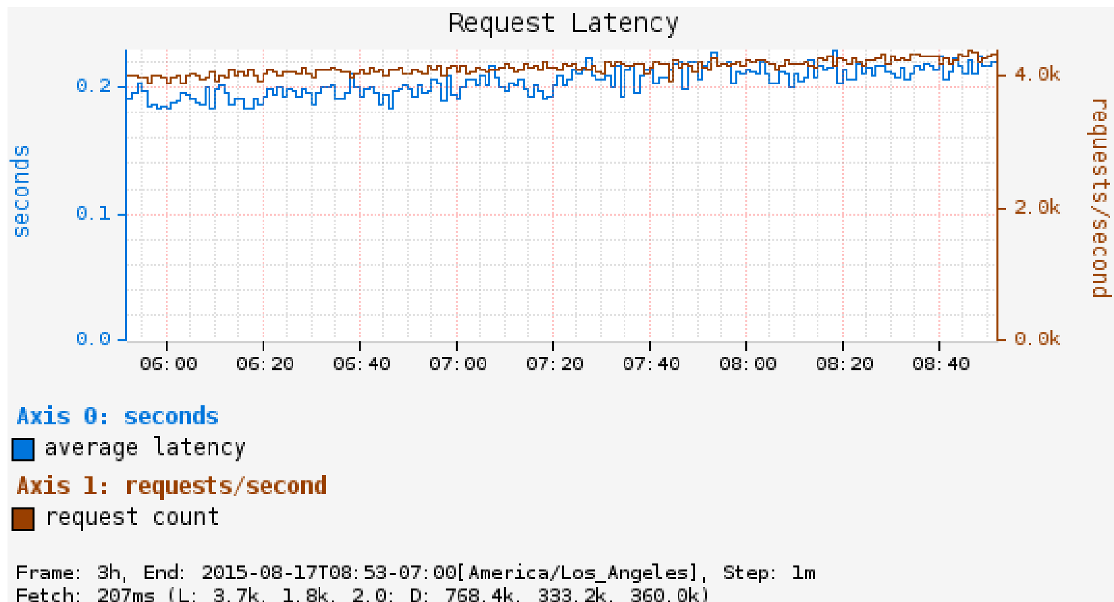


图 4.请求延迟

```
// create a timer with a name and a set of tags
Timer timer = registry.timer("timerName", "tagKey1",
"tagValue1", ...);

// execute an operation and time it at the same time
T result = timer.record(() -> fooReturnsT());

// alternatively, if you must manually record the time
Long start = System.nanoTime();
T result = fooReturnsT();
timer.record(System.nanoTime() - start,
TimeUnit.NANOSECONDS);
```

计时器同时记录 4 个统计信息：count，max，totalOfSquares 和 totalTime。如果您在每次记录时间时在计数器上调用了 `increment()` 一次，计数统计量将始终与计数器提供的单个归一化值相匹配，因此对于单个操作，不需要单独计数和分时。

对于[长时间运行的操作]，Spectator 提供了一个特殊的 `LongTaskTimer`。

Spectator 量规

量规用于确定一些当前值，如队列的大小或处于运行状态的线程数。由于仪表被采样，它们不提供关于这些值在样品之间如何波动的信息。

仪器的正常使用包括在初始化中使用标识符注册仪表，对要采样的对象的引用，以及基于对象获取或计算数值的功能。对对象的引用被单独传递，Spectator 注册表将保留对该对象的弱引用。如果对象被垃圾回收，则 Spectator 将自动删除注册。见注 Spectator 是关于潜在的内存泄漏的文件中，如果这个 API 被滥用。

```
// the registry will automatically sample this gauge
periodically
registry.gauge("gaugeName", pool,
Pool::numberOfRunningThreads);

// manually sample a value in code at periodic intervals
-- last resort!
registry.gauge("gaugeName", Arrays.asList("tagKey1",
"tagValue1", ...), 1000);
```

Spectator 分发摘要

分发摘要用于跟踪事件的分布情况。它类似于一个计时器，但更普遍的是，大小不一定是一段时间。例如，分发摘要可用于测量服务器的请求的有效载荷大小。

```
// the registry will automatically sample this gauge
periodically
DistributionSummary ds =
registry.distributionSummary("dsName", "tagKey1",
"tagValue1", ...);
ds.record(request.sizeInBytes());
```

指标集：Servo

警告	如果您的代码在 Java 8 上编译，请使用 Spectator 而不是 Servo，因为 Spectator

在 Servo 语言中，监视器是一个命名，键入和标记的配置，而指标表示给定监视器在某个时间点的值。Servo 显示器在逻辑上相当于 Spectator 米。Servo 显示器由 `MonitorRegistry` 创建和控制。尽管有上述警告，Servo 确实具有比 Spectator 有米的更广泛的监视器选项。

Spring Cloud 集成为您配置可注入的 `com.netflix.servo.MonitorRegistry` 实例。在 Servo 中创建了相应的 `Monitor` 类型后，记录数据的过程完全类似于 Spectator。

创建 Servo 显示器

如果您正在使用由 Spring Cloud 提供的 Servo `MonitorRegistry` 实例（具体来说是 `DefaultMonitorRegistry` 的实例），则 Servo 提供了用于检索计数器和计时器的便利类。这些便利类确保每个唯一的名称和标签组合只注册一个 `Monitor`。

要在 Servo 中手动创建监视器类型，特别是对于不提供方便方法的异域监视器类型，通过提供 `MonitorConfig` 实例来实例化适当的类型：

```
MonitorConfig config =
MonitorConfig.builder("timerName").withTag("tagKey1",
"tagValue1").build();

// somewhere we should cache this Monitor by
MonitorConfig
Timer timer = new BasicTimer(config);
monitorRegistry.register(timer);
```

指标后端：Atlas

Netflix 开发了 Atlas 来管理维度时间序列数据，实现近实时操作洞察。Atlas 具有内存中数据存储功能，可以非常快速地收集和报告大量的指标。

Atlas 捕获操作情报。而商业智能是收集的数据，用于分析一段时间内的趋势，操作情报提供了系统中目前发生的情况。

Spring Cloud 提供了一个 `spring-cloud-starter-atlas`，它具有您需要的所有依赖关系。然后只需使用`@EnableAtlas`注释您的 Spring Boot 应用程序，并为您运行的 Atlas 服务器提供 `netflix.atlas.uri` 属性的位置。

全球标签

您可以通过 Spring Cloud 向发送到 Atlas 后端的每个度量标准添加标签。全局标签可用于按应用程序名称，环境，区域等分隔度量。

实现 `AtlasTagProvider` 的每个 bean 将贡献全局标签列表：

```
@Bean
AtlasTagProvider atlasCommonTags(
    @Value("${spring.application.name}") String appName) {
  return () -> Collections.singletonMap("app", appName);
}
```

使用 Atlas

要引导内存独立的 Atlas 实例：

```
$ curl -LO
https://github.com/Netflix/atlas/releases/download/v1.4.2
/atlas-1.4.2-standalone.jar
$ java -jar atlas-1.4.2-standalone.jar
```

提示	运行在 r3.2xlarge（61GB RAM）上的 Atlas 独立节点可以在给定的 6 小时窗口内每

一旦运行，您收集了少量指标，请通过在 Atlas 服务器上列出代码来验证您的设置是否正确：

```
$ curl http://ATLAS/api/v1/tags
```

提示	在针对您的服务执行多个请求后，您可以通过在浏览器中粘贴以下 URL 来收集关于每 信息：http://ATLAS/api/v1/graph?q=name,rest,:eq,:avg

Atlas wiki 包含各种场景样本查询的汇编。

确保使用双指数平滑来查看警报原理和文档，以生成动态警报阈值。

重试失败的请求

Spring Cloud Netflix 提供了多种方式来进行 HTTP 请求。您可以使用负载平衡 `RestTemplate`，Ribbon 或 Feign。无论您如何选择 HTTP 请求，始终有可能失败的请求。当请求失败时，您可能希望自动重试该请求。要在使用 Sping Cloud Netflix 时完成此操作，您需要在应用程序的类路径中包含 Spring 重试。当 Spring 重试出现负载平衡 `RestTemplates` 时，Feign 和 Zuul 将自动重试任何失败的请求（假设配置允许）。

组态

随时 Ribbon 与 Spring 重试一起使用，您可以通过配置某些 Ribbon 属性来控制重试功能。您可以使用的属性是 `client.ribbon.MaxAutoRetries`，`client.ribbon.MaxAutoRetriesNextServer` 和

`client.ribbon.OkToRetryOnAllOperations`。请参阅 Ribbon 文档，了解属性的具体内容。

此外，您可能希望在响应中返回某些状态代码时重试请求。您可以列出您希望 Ribbon 客户端使用属性 `clientName.ribbon.retryableStatusCodes` 重试的响应代码。例如

```
clientName:
  ribbon:
    retryableStatusCodes: 404,502
```

您还可以创建一个类型为 `LoadBalancedRetryPolicy` 的 bean，并实现 `retryableStatusCode` 方法来确定是否要重试发出状态代码的请求。

Zuul

您可以通过将 `zuul.retryable` 设置为 `false` 来关闭 Zuul 的重试功能。您还可以通过将 `zuul.routes.routename.retryable` 设置为 `false`，以路由方式禁用重试功能。

Spring Cloud Stream

本节将详细介绍如何使用 Spring Cloud Stream。它涵盖了创建和运行流应用程序等主题。

介绍 Spring Cloud Stream

Spring Cloud Stream 是构建消息驱动的微服务应用程序的框架。Spring Cloud Stream 基于 Spring Boot 建立独立的生产级 Spring 应用程序，并使用 Spring Integration 提供与消息代理的连接。它提供了来自几家供应商的中间件的意见配置，介绍了持久发布订阅语义，消费者组和分区的概念。

您可以将`@EnableBinding`注释添加到应用程序，以便立即连接到消息代理，并且可以将`@StreamListener`添加到方法中，以使其接收流处理的事件。以下是接收外部消息的简单接收器应用程序。

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class VoteRecordingSinkApplication {

  public static void main(String[] args) {

SpringApplication.run(VoteRecordingSinkApplication.class,
args);
  }

  @StreamListener(Sink.INPUT)
  public void processVote(Vote vote) {
      votingService.recordVote(vote);
  }
}
```

`@EnableBinding`注释需要一个或多个接口作为参数（在这种情况下，该参数是单个 `Sink` 接口）。接口声明输入和/或输出通道。Spring Cloud Stream 提供了接口 `Source`，`Sink` 和 `Processor`; 您还可以定义自己的界面。

以下是 `Sink` 接口的定义：

```
public interface Sink {
  String INPUT = "input";

```

```
  @Input(Sink.INPUT)
  SubscribableChannel input();
}
```

`@Input` 注释标识*输入通道*，通过该*输入通道*接收到的消息进入应用程序; `@Output` 注释标识*输出通道*，发布的消息将通过该*通道*离开应用程序。

`@Input` 和`@Output` 注释可以使用频道名称作为参数; 如果未提供名称，将使用注释方法的名称。

Spring Cloud Stream 将为您创建一个界面的实现。您可以在应用程序中通过自动连接来使用它，如下面的测试用例示例。

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes =
VoteRecordingSinkApplication.class)
@WebAppConfiguration
@DirtiesContext
public class StreamApplicationTests {

  @Autowired
  private Sink sink;

  @Test
  public void contextLoads() {
    assertNotNull(this.sink.input());
  }
}
```

主要概念

Spring Cloud Stream 提供了一些简化了消息驱动的微服务应用程序编写的抽象和原语。本节概述了以下内容:

- Spring Cloud Stream 的应用模型
- Binder 抽象
- 持续的发布 - 订阅支持
- 消费者群体支持
- 分区支持
- 一个可插拔的 Binder API

应用模型

一个 Spring Cloud Stream 应用程序由一个中间件中立的核心组成。该应用程序通过 Spring Cloud Stream 注入到其中的输入和输出*通道*与外界进行通信。渠道通过中间件特定的 Binder 实现连接到外部经纪人。

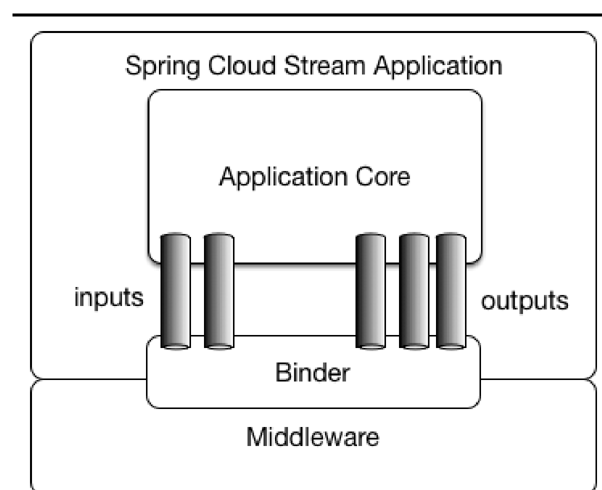


图 5. Spring Cloud Stream 应用

胖 JAR

Spring Cloud Stream 应用程序可以在独立模式下从 IDE 运行进行测试。要在生产中运行 Spring Cloud Stream 应用程序，您可以使用为 Maven 或 Gradle 提供的标准 Spring Boot 工具创建可执行文件（或“胖”）JAR。

Binder 抽象

Spring Cloud Stream 为 Kafka 和 Rabbit MQ 提供 Binder 实现。Spring Cloud Stream 还包括一个 TestSupportBinder，它保留了一个未修改的通道，以便测试可以直接和可靠地与通道进行交互。您可以使用可扩展 API 编写自己的 Binder。

Spring Cloud Stream 使用 Spring Boot 进行配置，Binder 抽象使得 Spring Cloud Stream 应用程序可以灵活地连接到中间件。例如，部署者可以在运行时动态地选择通道连接的目的地（例如，Kafka 主题或 RabbitMQ 交换）。可以通过外部配置属性和 Spring Boot（包括应用程序参数，环境变量和 `application.yml` 或 `application.properties` 文件）支持的任何形式提供此类配置。在引入 Spring Cloud Stream 部分的接收器示例中，将应用程序属性 `spring.cloud.stream.bindings.input.destination` 设置为 `raw-sensor-data` 将使其从 `raw-sensor-data` Kafka 主题或从绑定到 `raw-sensor-data` RabbitMQ 交换。

Spring Cloud Stream 自动检测并使用类路径中找到的 binder。您可以使用相同的代码轻松使用不同类型的中间件：在构建时只包含不同的绑定器。对于更复杂的用例，您还可以在应用程序中打包多个绑定器，并在运行时选择绑定器，甚至是否为不同的通道使用不同的绑定器。

持续发布 - 订阅支持

应用之间的通信遵循发布订阅模式，其中通过共享主题广播数据。这可以在下图中看到，它显示了一组交互式的 Spring Cloud Stream 应用程序的典型部署。

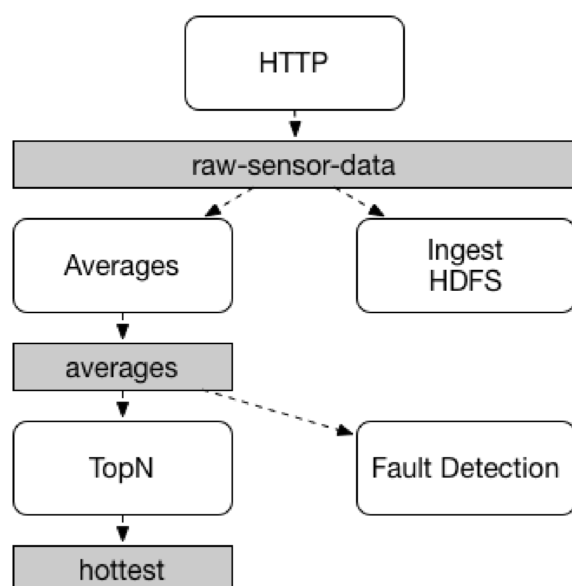


图 6. Spring Cloud Stream Publish-Subscribe

传感器向 HTTP 端点报告的数据将发送到名为 `raw-sensor-data` 的公共目标。从目的地，它由微服务应用程序独立处理，该应用程序计算时间窗口平均值，以及另一个将原始数据导入 HDFS 的微服务应用程序。为了处理数据，两个应用程序在运行时将主题声明为它们的输入。

发布订阅通信模型降低了生产者和消费者的复杂性，并允许将新应用程序添加到拓扑中，而不会中断现有流。例如，在平均计算应用程序的下游，您可以添加一个计算显示和监视的最高温度值的应用程序。然后，您可以添加另一个解释相同

的故障检测平均流程的应用程序。通过共享主题而不是点对点队列进行所有通信可以减少微服务之间的耦合。

虽然发布订阅消息的概念不是新的，但是 Spring Cloud Stream 需要额外的步骤才能使其成为其应用模型的一个有意义的选择。通过使用本地中间件支持，Spring Cloud Stream 还简化了在不同平台上使用发布订阅模型。

消费群体

虽然发布订阅模型可以轻松地通过共享主题连接应用程序，但通过创建给定应用程序的多个实例来扩展的能力同样重要。当这样做时，应用程序的不同实例被放置在竞争的消费者关系中，其中只有一个实例预期处理给定消息。

Spring Cloud Stream 通过*消费者组*的概念来模拟此行为。（Spring Cloud Stream 消费者组与 Kafka 消费者组相似并受到启发。）每个消费者绑定可以使用 `spring.cloud.stream.bindings.<channelName>.group` 属性来指定组名称。对于下图所示的消费者，此属性将设置为 `spring.cloud.stream.bindings.<channelName>.group=hdfsWrite` 或 `spring.cloud.stream.bindings.<channelName>.group=average`。

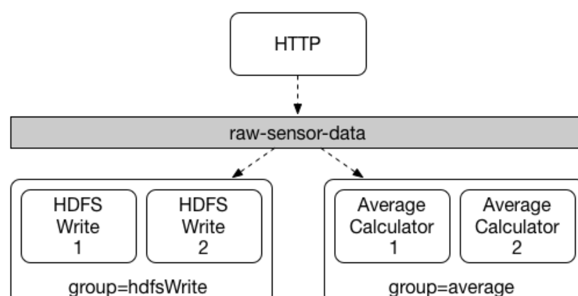


图 7. Spring Cloud Stream 消费者组

订阅给定目标的所有组都会收到已发布数据的副本，但每个组中只有一个成员从该目的地接收给定的消息。默认情况下，当未指定组时，Spring Cloud Stream 将应用程序分配给与所有其他消费者组发布 - 订阅关系的匿名独立单个成员消费者组。

耐久力

符合 Spring Cloud Stream 的有意义的应用模式，消费者群体订阅是*持久的*。也就是说，绑定实现确保组预订是持久的，一旦已经创建了一个组的至少一个订阅，即使组中的所有应用程序都被停止，组也将接收消息。

注意	匿名订阅本质上是不耐用的。对于某些 binder 实现（例如 RabbitMQ），可以具有非

通常，当将应用绑定到给定目的地时，最好始终指定消费者组。在扩展 Spring Cloud Stream 应用程序时，必须为每个输入绑定指定一个使用者组。这样可以防止应用程序的实例收到重复的消息（除非需要这种行为，这是不寻常的）。

分区支持

Spring Cloud Stream 提供对给定应用程序的多个实例之间的*分区*数据的支持。在分区场景中，物理通信介质（例如，代理主题）被视为被构造成多个分区。一个或多个生产者应用程序实例将数据发送到多个消费者应用程序实例，并确保由共同特征标识的数据由相同的消费者实例处理。

Spring Cloud Stream 提供了统一方式实现分区处理用例的通用抽象。因此，无论代理本身是否自然分区（例如 Kafka）（例如 RabbitMQ），分区可以被使用。

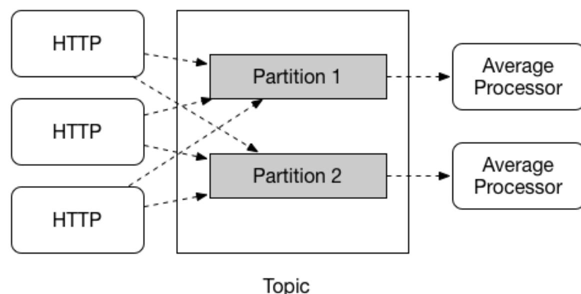


图 8. Spring Cloud Stream 分区

分区是状态处理中的一个关键概念，无论是性能还是一致性原因，它都是批评性的，以确保所有相关数据一起处理。例如，在时间平均计算示例中，重要的是所有给定传感器的所有测量都由相同的应用实例进行处理。

注意	要设置分区处理方案，您必须同时配置数据生成和数据消耗结束。

编程模型

本节介绍 Spring Cloud Stream 的编程模型。Spring Cloud Stream 提供了许多预定义的注释，用于声明绑定的输入和输出通道，以及如何收听频道。

声明和绑定频道

触发绑定`@EnableBinding`

您可以将 Spring 应用程序转换为 Spring Cloud Stream 应用程序，将 `@EnableBinding` 注释应用于应用程序的配置类之一。`@EnableBinding` 注释本身使用`@Configuration` 进行元注释，并触发 Spring Cloud Stream 基础架构的配置：

```
...
@Import(...)
@Configuration
@EnableIntegration
public @interface EnableBinding {
    ...
    Class<?>[] value() default {};
}
```

`@EnableBinding` 注释可以将一个或多个接口类作为参数，这些接口类包含表示可绑定组件（通常是消息通道）的方法。

注意	在 Spring Cloud Stream 1.0 中，唯一支持的可绑定组件是 Spring 消息传递 `Message` `SubscribableChannel` 和 `PollableChannel`。未来版本应该使用相同的机制将档中，我们将继续参考渠道。

`@Input` 和`@Output`

Spring Cloud Stream 应用程序可以在接口中定义任意数量的输入和输出通道为 `@Input` 和`@Output` 方法：

```
public interface Barista {

    @Input
    SubscribableChannel orders();

    @Output
    MessageChannel hotDrinks();

    @Output
```

```
    MessageChannel coldDrinks();
}
```

使用此接口作为参数`@EnableBinding`将分别触发三个绑定的通道名称为 `orders`，`hotDrinks`和`coldDrinks`。

```
@EnableBinding(Barista.class)
public class CafeConfiguration {

   ...
}
```

自定义频道名称

使用`@Input`和`@Output`注释，您可以指定频道的自定义频道名称，如以下示例所示：

```
public interface Barista {
    ...
    @Input("inboundOrders")
    SubscribableChannel orders();
}
```

在这个例子中，创建的绑定通道将被命名为`inboundOrders`。

`Source`，`Sink`和`Processor`

为了方便寻址最常见的用例，涉及输入通道，输出通道或两者，Spring Cloud Stream 提供了开箱即用的三个预定义接口。

`Source`可用于具有单个出站通道的应用程序。

```
public interface Source {

  String OUTPUT = "output";
```

```
  @Output(Source.OUTPUT)
  MessageChannel output();

}
```

`Sink` 可用于具有单个入站通道的应用程序。

```
public interface Sink {

  String INPUT = "input";

  @Input(Sink.INPUT)
  SubscribableChannel input();

}
```

`Processor` 可用于具有入站通道和出站通道的应用程序。

```
public interface Processor extends Source, Sink {
}
```

Spring Cloud Stream 不为任何这些接口提供特殊处理; 它们只是开箱即用。

访问绑定通道

注入绑定界面

对于每个绑定接口，Spring Cloud Stream 将生成一个实现该接口的 bean。调用其中一个 bean 的`@Input` 注释或`@Output` 注释方法将返回相关的绑定通道。

以下示例中的 bean 在调用其 `hello` 方法时在输出通道上发送消息。它在注入的 `Source` bean 上调用 `output()`来检索目标通道。

```
@Component
public class SendingBean {
```

```
    private Source source;

    @Autowired
    public SendingBean(Source source) {
        this.source = source;
    }

    public void sayHello(String name) {

source.output().send(MessageBuilder.withPayload(name).bui
ld());
    }
}
```

直接注入渠道

绑定通道也可以直接注入:

```
@Component
public class SendingBean {

    private MessageChannel output;

    @Autowired
    public SendingBean(MessageChannel output) {
        this.output = output;
    }

    public void sayHello(String name) {

output.send(MessageBuilder.withPayload(name).build());
    }
}
```

如果在声明注释上定制了通道的名称，则应使用该名称而不是方法名称。给出以下声明:

```
public interface CustomSource {
    ...
    @Output("customOutput")
```

```
    MessageChannel output();
}
```

通道将被注入，如下例所示：

```
@Component
public class SendingBean {

    private MessageChannel output;

    @Autowired
    public SendingBean(@Qualifier("customOutput")
MessageChannel output) {
        this.output = output;
    }

    public void sayHello(String name) {

this.output.send(MessageBuilder.withPayload(name).build()
);
    }
}
```

生产和消费消息

您可以使用 Spring Integration 注解或 Spring Cloud Stream 的 `@StreamListener` 注释编写 Spring Cloud Stream 应用程序。`@StreamListener` 注释在其他 Spring 消息传递注释（例如 `@MessageMapping`，`@JmsListener`，`@RabbitListener` 等）之后建模，但添加内容类型管理和类型强制功能。

本地 Spring Integration 支持

由于 Spring Cloud Stream 基于 Spring Integration，Stream 完全继承了 Integration 的基础和基础架构以及组件本身。例如，您可以将 `Source` 的输出通道附加到 `MessageSource`：

```
@EnableBinding(Source.class)
public class TimerSource {

  @Value("${format}")
  private String format;

  @Bean
  @InboundChannelAdapter(value = Source.OUTPUT, poller =
@Poller(fixedDelay = "${fixedDelay}", maxMessagesPerPoll
= "1"))
  public MessageSource<String> timerMessageSource() {
    return () -> new GenericMessage<>(new
SimpleDateFormat(format).format(new Date()));
  }
}
```

或者您可以在变压器中使用处理器的通道：

```
@EnableBinding(Processor.class)
public class TransformProcessor {
  @Transformer(inputChannel = Processor.INPUT,
outputChannel = Processor.OUTPUT)
  public Object transform(String message) {
    return message.toUpperCase();
  }
}
```

Spring Integration 错误通道支持

Spring Cloud Stream 支持发布 Spring Integration 全局错误通道收到的错误消息。发送到 `errorChannel` 的错误消息可以通过为名为 `error` 的出站目标配置绑定，将其发布到代理的特定目标。例如，要将错误消息发布到名为“myErrors”

的代理目标，请提供以下属性：`spring.cloud.stream.bindings.error.destination=myErrors`

使用@StreamListener 进行自动内容类型处理

Spring Integration 支持 Spring Cloud Stream 提供自己的`@StreamListener`注释，以其他 Spring 消息传递注释（例如`@MessageMapping`，`@JmsListener`，`@RabbitListener`等））。`@StreamListener`注释提供了一种更简单的处理入站邮件的模型，特别是在处理涉及内容类型管理和类型强制的用例时。

Spring Cloud Stream 提供了一种可扩展的`MessageConverter`机制，用于通过绑定通道处理数据转换，并且在这种情况下，将调度到使用`@StreamListener`注释的方法。以下是处理外部`Vote`事件的应用程序的示例：

```
@EnableBinding(Sink.class)
public class VoteHandler {

  @Autowired
  VotingService votingService;

  @StreamListener(Sink.INPUT)
  public void handle(Vote vote) {
    votingService.record(vote);
  }
}
```

当考虑`String`有效载荷和`contentType`标题`application/json`的入站`Message`时，可以看到`@StreamListener`和 Spring Integration `@ServiceActivator`之间的区别。在`@StreamListener`的情况下，`MessageConverter`机制将使用`contentType`标头将`String`有效载荷解析为`Vote`对象。

与其他 Spring 消息传递方法一样，方法参数可以用`@Payload`，`@Headers`和`@Header`注释。

注意	对于返回数据的方法，您必须使用`@SendTo`注释来指定方法返回的数据的输出绑定

```
@EnableBinding(Processor.class)
public class TransformProcessor {

  @Autowired
  VotingService votingService;

  @StreamListener(Processor.INPUT)
  @SendTo(Processor.OUTPUT)
  public VoteResult handle(Vote vote) {
    return votingService.record(vote);
  }
}
```

使用@StreamListener 将消息分派到多个方法

自 1.2 版本以来，Spring Cloud Stream 支持根据条件向在输入通道上注册的多个`@StreamListener`方法发送消息。

为了有资格支持有条件的调度，一种方法必须满足以下条件：

- 它不能返回值
- 它必须是一个单独的消息处理方法（不支持的反应 API 方法）

条件通过注释的`condition`属性中的 SpEL 表达式指定，并为每个消息进行评估。匹配条件的所有处理程序将在同一个线程中被调用，并且不必对调用发生的顺序做出假设。

使用@StreamListener具有调度条件的示例可以在下面看到。在此示例中，带有值为 foo 的标题 type 的所有消息将被分派到 receiveFoo 方法，所有带有值为 bar 的标题 type 的消息将被分派到 receiveBar 方法。

```
@EnableBinding(Sink.class)
@EnableAutoConfiguration
public static class TestPojoWithAnnotatedArguments {

    @StreamListener(target = Sink.INPUT, condition =
"headers['type']=='foo'")
    public void receiveFoo(@Payload FooPojo fooPojo) {
       // handle the message
    }

    @StreamListener(target = Sink.INPUT, condition =
"headers['type']=='bar'")
    public void receiveBar(@Payload BarPojo barPojo) {
       // handle the message
    }
}
```

注意	仅通过@StreamListener条件进行调度仅对单个消息的处理程序支持，而不适用于

反应式编程支持

Spring Cloud Stream 还支持使用反应性 API，其中将传入和传出数据作为连续数据流处理。通过 spring-cloud-stream-reactive 提供对反应性 API 的支持，需要将其明确添加到您的项目中。

具有反应性 API 的编程模型是声明式的，而不是指定如何处理每个单独的消息，您可以使用描述从入站到出站数据流的功能转换的运算符。

Spring Cloud Stream 支持以下反应性 API：

- 反应堆
- RxJava 1.x

将来，它旨在支持基于活动流的更通用的模型。

反应式编程模型还使用`@StreamListener`注释来设置反应处理程序。差异在于：

- `@StreamListener`注释不能指定输入或输出，因为它们作为参数提供，并从方法返回值；
- 必须使用`@Input`和`@Output`注释方法的参数，指示输入和分别输出的数据流连接到哪个输入或输出；
- 方法的返回值（如果有的话）将用`@Output`注释，表示要发送数据的输入。

注意	反应式编程支持需要 Java 1.8。
注意	截至 Spring Cloud Stream 1.1.1 及更高版本（从布鲁克林发行版开始列出），反应式 3.0.4.RELEASE 和更高版本。不支持早期的 Reactor 版本（包括 3.0.1.RELEASE，3.0 `spring-cloud-stream-reactive` 将会过渡地检索正确的版本，但项目结构有可 `io.projectreactor:reactor-core` 的版本管理到较早版本，特别是在使用 Ma （Spring Boot 1.x）生成的项目，这将覆盖 Reactor 版本为 `2.0.8.RELEASE`。在这 工件。这可以通过在 `io.projectreactor:reactor-core` 上直接依赖于 `3.0.4` 实现。
注意	术语 `reactive` 的使用目前指的是正在使用的反应性 API，而不是执行模型是无效的 “拉”模型）。虽然通过使用 Reactor 提供了一些背压支持，但我们希望长期以来通过 支持完全无反应的管道。

基于反应器的处理程序

基于反应器的处理程序可以具有以下参数类型：

- 对于用`@Input`注释的参数，它支持反应器类型 `Flux`。入站通量的参数化遵循与单个消息处理相同的规则：它可以是整个 `Message`，一个可以是 `Message` 有效负载的 POJO，也可以是一个 POJO 基于 `Message` 内容类型头的转换。提供多个输入;
- 对于使用 `Output` 注释的参数，它支持将方法生成的 `Flux` 与输出连接的类型 `FluxSender`。一般来说，仅当方法可以有多个输出时才建议指定输出作为参数;

基于反应器的处理程序支持 `Flux` 的返回类型，其中必须使用`@Output` 注释。当单个输出通量可用时，我们建议使用该方法的返回值。

这是一个简单的基于反应器的处理器的例子。

```
@EnableBinding(Processor.class)
@EnableAutoConfiguration
public static class UppercaseTransformer {

  @StreamListener
  @Output(Processor.OUTPUT)
  public Flux<String> receive(@Input(Processor.INPUT)
Flux<String> input) {
    return input.map(s -> s.toUpperCase());
  }
}
```

使用输出参数的同一个处理器如下所示：

```
@EnableBinding(Processor.class)
@EnableAutoConfiguration
public static class UppercaseTransformer {

  @StreamListener
```

```
  public void receive(@Input(Processor.INPUT) Flux<String>
input,
     @Output(Processor.OUTPUT) FluxSender output) {
     output.send(input.map(s -> s.toUpperCase()));
  }
}
```

RxJava 1.x 支持

RxJava 1.x 处理程序遵循与基于反应器的规则相同的规则，但将使用 `Observable` 和 `ObservableSender` 参数和返回类型。

所以上面的第一个例子会变成:

```
@EnableBinding(Processor.class)
@EnableAutoConfiguration
public static class UppercaseTransformer {

  @StreamListener
  @Output(Processor.OUTPUT)
  public Observable<String>
receive(@Input(Processor.INPUT) Observable<String> input)
{
    return input.map(s -> s.toUpperCase());
  }
}
```

上面的第二个例子将会变成:

```
@EnableBinding(Processor.class)
@EnableAutoConfiguration
public static class UppercaseTransformer {

  @StreamListener
  public void receive(@Input(Processor.INPUT)
Observable<String> input,
     @Output(Processor.OUTPUT) ObservableSender output) {
     output.send(input.map(s -> s.toUpperCase()));
  }
}
```

聚合

Spring Cloud Stream 支持将多个应用程序聚合在一起，直接连接其输入和输出通道，并避免通过代理交换消息的额外成本。从版本 1.0 的 Spring Cloud Stream 开始，仅对以下类型的应用程序支持聚合：

- *来源* - 具有名为 `output` 的单个输出通道的应用程序，通常具有类型为 `org.springframework.cloud.stream.messaging.Source` 的单个绑定
- *接收器* - 具有名为 `input` 的单个输入通道的应用程序，通常具有类型为 `org.springframework.cloud.stream.messaging.Sink` 的单个绑定
- *处理器* - 具有名为 `input` 的单个输入通道和名为 `output` 的单个输出通道的应用程序，通常具有类型为 `org.springframework.cloud.stream.messaging.Processor` 的单个绑定。

它们可以通过创建一系列互连的应用程序来聚合在一起，其中序列中的元素的输出通道连接到下一个元素的输入通道（如果存在）。序列可以从*源*或*处理器开始*，它可以包含任意数量的*处理器*，并且必须以*处理器*或*接收器*结束。

根据起始和结束元素的性质，序列可以具有一个或多个可绑定的信道，如下所示：

- 如果序列从源头开始并以 sink 结束，则应用程序之间的所有通信都是直接的，并且不会绑定任何通道
- 如果序列以处理器开始，则其输入通道将成为聚合的 `input` 通道，并将相应地进行绑定
- 如果序列以处理器结束，则其输出通道将成为聚合的 `output` 通道，并将相应地进行绑定

使用 `AggregateApplicationBuilder` 实用程序类执行聚合，如以下示例所示。我们考虑一个项目，我们有源，处理器和汇点，可以在项目中定义，或者可以包含在项目的依赖之一中。

注意	如果配置类使用`@SpringBootApplication`，聚合应用程序中的每个组件（源，行 这是为了避免应用程序之间的串扰，由于在同一个包内的配置类上由`@SpringBoot` 下面的示例中，可以看到，Source，Processor 和 Sink 应用程序类分组在单独的包中 `@Configuration` 类中提供源，宿或处理器配置，避免使用`@SpringBootApplic` 进行聚合。

```
package com.app.mysink;

@SpringBootApplication
@EnableBinding(Sink.class)
public class SinkApplication {

	private static Logger logger =
LoggerFactory.getLogger(SinkApplication.class);

	@ServiceActivator(inputChannel=Sink.INPUT)
	public void loggerSink(Object payload) {
		logger.info("Received: " + payload);
	}
}
package com.app.myprocessor;

@SpringBootApplication
```

```
@EnableBinding(Processor.class)
public class ProcessorApplication {

	@Transformer
	public String loggerSink(String payload) {
		return payload.toUpperCase();
	}
}
package com.app.mysource;

@SpringBootApplication
@EnableBinding(Source.class)
public class SourceApplication {

	@Bean
	@InboundChannelAdapter(value = Source.OUTPUT)
	public String timerMessageSource() {
		return new SimpleDateFormat().format(new
Date());
	}
}
```

每个配置可以用于运行一个单独的组件，但在这种情况下，它们可以聚合在一起，如下所示：

```
package com.app;

@SpringBootApplication
public class SampleAggregateApplication {

	public static void main(String[] args) {
		new AggregateApplicationBuilder()
			.from(SourceApplication.class).args("-
-fixedDelay=5000")
			.via(ProcessorApplication.class)
			.to(SinkApplication.class).args("--
debug=true").run(args);
	}
}
```

该序列的起始组件作为 `from()` 方法的参数提供。序列的结尾部分作为 `to()` 方法的参数提供。中间处理器作为 `via()` 方法的参数提供。同一类型的多个处理器可以一起链接（例如，用于具有不同配置的流水线转换）。对于每个组件，构建器可以为 Spring Boot 配置提供运行时参数。

配置聚合应用程序

Spring Cloud Stream 支持使用'namespace'作为前缀向聚合应用程序内的各个应用程序传递属性。

命名空间可以为应用程序设置如下:

```
@SpringBootApplication
public class SampleAggregateApplication {

	public static void main(String[] args) {
		new AggregateApplicationBuilder()

	.from(SourceApplication.class).namespace("source").
args("--fixedDelay=5000")

	.via(ProcessorApplication.class).namespace("process
or1")

	.to(SinkApplication.class).namespace("sink").args("
--debug=true").run(args);
	}
}
```

一旦为单个应用程序设置了“命名空间”，则可以使用任何支持的属性源（命令行，环境属性等）将具有 `namespace` 作为前缀的应用程序属性传递到聚合应用程序，

例如，要覆盖“source”和“sink”应用程序的默认 `fixedDelay` 和 `debug` 属性：

```
java -jar target/MyAggregateApplication-0.0.1-
SNAPSHOT.jar --source.fixedDelay=10000 --sink.debug=false
```

为非自包含聚合应用程序配置绑定服务属性

非自包含聚合应用程序通过聚合应用程序的入站/出站组件（通常为消息通道）中的一个或两者绑定到外部代理，而聚合应用程序内的应用程序是直接绑定的。例如：源应用程序的输出和处理器应用程序的输入是直接绑定的，而处理器的输出通道绑定到代理的外部目的地。当传递非自包含聚合应用程序的绑定服务属性时，需要将绑定服务属性传递给聚合应用程序，而不是将它们设置为单个子应用程序的“args”。例如，

```
@SpringBootApplication
public class SampleAggregateApplication {

      public static void main(String[] args) {
            new AggregateApplicationBuilder()

      .from(SourceApplication.class).namespace("source").
args("--fixedDelay=5000")

      .via(ProcessorApplication.class).namespace("process
or1").args("--debug=true").run(args);
      }
}
```

需要将绑定属性-- `spring.cloud.stream.bindings.output.destination=processor-output` 指定为外部配置属性（cmdline arg 等）之一。

Binders

Spring Cloud Stream 提供了一个 Binder 抽象，用于连接到外部中间件的物理目标。本节提供有关 Binder SPI，其主要组件和实现特定详细信息背后的主要概念的信息。

生产者和消费者

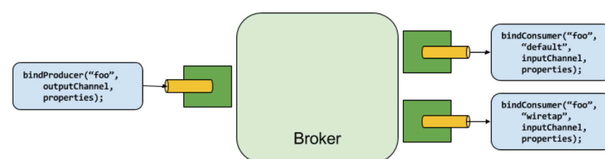


图 9.生产者和消费者

甲*生产者*是将消息发送到信道的任何组分。该通道可以通过该代理的 Binder 实现绑定到外部消息代理。当调用 `bindProducer()` 方法时，第一个参数是代理中目标的名称，第二个参数是生成器将发送消息的本地通道实例，第三个参数包含属性（如分区键表达式）在为该通道创建的适配器中使用。

甲*消费者*的是，从一个信道接收的消息的任何组分。与生产者一样，消费者的频道可以绑定到外部消息代理。当调用 `bindConsumer()` 方法时，第一个参数是目标名称，第二个参数提供消费者逻辑组的名称。由给定目的地的消费者绑定表示的每个组接收生产者发送到该目的地的每个消息的副本（即，发布 - 订阅语义）。如果有多个使用相同组名称的消费者实例绑定，那么消息将在这些消费者实例之间进行负载平衡，以便生产者发送的每个消息仅由每个组中的单个消费者实例消耗（即排队语义）。

Binder SPI

Binder SPI 包括许多接口，即插即用实用程序类和发现策略，提供可插拔机制连接到外部中间件。

SPI 的关键点是`Binder`接口，它是将输入和输出连接到外部中间件的策略。

```
public interface Binder<T, C extends ConsumerProperties,
P extends ProducerProperties> {
	Binding<T> bindConsumer(String name, String group,
T inboundBindTarget, C consumerProperties);

	Binding<T> bindProducer(String name, T
outboundBindTarget, P producerProperties);
}
```

界面参数化，提供多个扩展点：

- 输入和输出绑定目标 - 从版本 1.0 开始，只支持`MessageChannel`，但是这个目标是将来用作扩展点;
- 扩展的消费者和生产者属性 - 允许特定的 Binder 实现来添加可以以类型安全的方式支持的补充属性。

典型的绑定实现包括以下内容

- 一个实现`Binder`接口的类;
- 一个 Spring `@Configuration`类，与中间件连接基础架构一起创建上述类型的 bean;
- 在类路径中找到的包含一个或多个绑定器定义的`META-INF/spring.binders`文件，例如

```
kafka:\
```

```
org.springframework.cloud.stream.binder.kafka.config.Kafk
aBinderConfiguration
```

Binder 检测

Spring Cloud Stream 依赖于 Binder SPI 的实现来执行将通道连接到消息代理的任务。每个 Binder 实现通常连接到一种类型的消息系统。

类路径检测

默认情况下，Spring Cloud Stream 依赖于 Spring Boot 的自动配置来配置绑定过程。如果在类路径中找到单个 Binder 实现，则 Spring Cloud Stream 将自动使用。例如，一个旨在绑定到 RabbitMQ 的 Spring Cloud Stream 项目可以简单地添加以下依赖项：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-
rabbit</artifactId>
</dependency>
```

对于其他绑定依赖关系的特定 maven 坐标，请参阅该 binder 实现的文档。

Classpath 上有多个 Binders

当类路径中存在多个绑定器时，应用程序必须指明每个通道绑定将使用哪个绑定器。每个 binder 配置都包含一个 `META-INF/spring.binders`，它是一个简单的属性文件：

```
rabbit:\
```

```
org.springframework.cloud.stream.binder.rabbit.config.RabbitServiceAutoConfiguration
```

对于其他提供的绑定实现（例如 Kafka），存在类似的文件，并且预期自定义绑定实现也可以提供它们。键代表 binder 实现的标识名称，而该值是以逗号分隔的配置类列表，每个配置类都包含唯一的一个类型为 `org.springframework.cloud.stream.binder.Binder` 的 bean 定义。

可以使用 `spring.cloud.stream.defaultBinder` 属性（例如 `spring.cloud.stream.defaultBinder=rabbit`）全局执行 Binder 选择，或通过在每个通道绑定上配置 binder 来单独执行。例如，从 Kafka 读取并写入 RabbitMQ 的处理器应用程序（分别具有用于读/写的名称为 `input` 和 `output` 的通道）可以指定以下配置：

```
spring.cloud.stream.bindings.input.binder=kafka
spring.cloud.stream.bindings.output.binder=rabbit
```

连接到多个系统

默认情况下，绑定器共享应用程序的 Spring Boot 自动配置，以便在类路径中找到每个绑定器的一个实例。如果您的应用程序连接到同一类型的多个代理，则可以指定多个绑定器配置，每个具有不同的环境设置。

注意	打开显式绑定器配置将完全禁用默认绑定器配置过程。如果这样做，所有使用的绑定用 Spring Cloud Stream 的框架可能会创建可以通过名称引用的 binder 配置，但不会配置可能将其 `defaultCandidate` 标志设置为 false，例如 `spring.cloud.stream.binders.<configurationName>.defaultCandida` binder 配置过程存在的配置。

例如，这是连接到两个 RabbitMQ 代理实例的处理器应用程序的典型配置：

```
spring:
  cloud:
    stream:
      bindings:
        input:
          destination: foo
          binder: rabbit1
        output:
          destination: bar
          binder: rabbit2
      binders:
        rabbit1:
          type: rabbit
          environment:
            spring:
              rabbitmq:
                host: <host1>
        rabbit2:
          type: rabbit
          environment:
            spring:
              rabbitmq:
                host: <host2>
```

Binder 配置属性

创建自定义绑定器配置时，以下属性可用。它们必须以

spring.cloud.stream.binders.<configurationName>为前缀。

类型

粘合剂类型。它通常引用在类路径中找到的绑定器之一，特别是 META-INF/spring.binders 文件中的键。

默认情况下，它具有与配置名称相同的值。

inheritEnvironment

配置是否会继承应用程序本身的环境。

默认 `true`。

环境

一组可用于自定义绑定环境的属性的根。配置此配置后，创建绑定器的上下文不是应用程序上下文的子级。这允许粘合剂组分和应用组分之间的完全分离。

默认 `empty`。

defaultCandidate

粘合剂配置是否被认为是默认的粘合剂的候选者，或者仅在明确引用时才能使用。这允许添加 binder 配置，而不会干扰默认处理。

默认 `true`。

配置选项

Spring Cloud Stream 支持常规配置选项以及绑定和绑定器的配置。一些绑定器允许额外的绑定属性来支持中间件特定的功能。

可以通过 Spring Boot 支持的任何机制将配置选项提供给 Spring Cloud Stream 应用程序。这包括应用程序参数，环境变量和 YAML 或.properties 文件。

Spring Cloud Stream Properties

spring.cloud.stream.instanceCount

应用程序部署实例的数量。必须设置分区，如果使用 Kafka。

默认值：`1`。

spring.cloud.stream.instanceIndex

应用程序的实例索引：从 `0` 到 `instanceCount` - 1 的数字。用于分区和使用 Kafka。在 Cloud Foundry 中自动设置以匹配应用程序的实例索引。

spring.cloud.stream.dynamicDestinations

可以动态绑定的目标列表（例如，在动态路由方案中）。如果设置，只能列出目的地。

默认值：空（允许任何目的地绑定）。

spring.cloud.stream.defaultBinder

如果配置了多个绑定器，则使用默认的 binder。请参阅 Classpath 上的 Multiple Binders。

默认值：空。

spring.cloud.stream.overrideCloudConnectors

此属性仅适用于 `cloud` 配置文件激活且 Spring Cloud 连接器随应用程序一起提供。如果属性为 false（默认值），绑定器将检测适合的绑定服务（例如，在 Cloud Foundry 中为 RabbitMQ 绑定器绑定的 RabbitMQ 服务），并将使用它来创建连接（通常通过 Spring Cloud 连接器）。当设置为 true 时，此属性指示绑定器完全忽略绑定的服务，并依赖 Spring Boot 属性（例如，依赖于 RabbitMQ 绑定器环境中提供的 `spring.rabbitmq.*` 属性）。当连接到多个系统时，此属性的典型用法将嵌套在定制环境中。

默认值：false。

绑定 Properties

绑定属性使用格式 `spring.cloud.stream.bindings.<channelName>.<property>=<value>` 提供。`<channelName>` 表示正在配置的通道的名称（例如 `Source` 的 `output`）。

为了避免重复，Spring Cloud Stream 支持所有通道的设置值，格式为 `spring.cloud.stream.default.<property>=<value>`。

在下面的内容中，我们指出我们在哪里省略了 `spring.cloud.stream.bindings.<channelName>.` 前缀，并且只关注属性名称，但有一个理解，前缀将被包含在运行时。

Properties 使用 Spring Cloud Stream

以下绑定属性可用于输入和输出绑定，并且必须以 `spring.cloud.stream.bindings.<channelName>.`为前缀，例如 `spring.cloud.stream.bindings.input.destination=ticktock`。

可以使用前缀 `spring.cloud.stream.default` 设置默认值，例如 `spring.cloud.stream.default.contentType=application/json`。

目的地

绑定中间件上的通道的目标目标（例如，RabbitMQ 交换或 Kafka 主题）。如果通道绑定为消费者，则可以将其绑定到多个目标，并且目标名称可以指定为逗号分隔的字符串值。如果未设置，则使用通道名称。此属性的默认值不能被覆盖。

组

渠道的消费群体。仅适用于入站绑定。参见消费者群体。

默认值：null（表示匿名消费者）。

内容类型

频道的内容类型。

默认值：null（以便不执行类型强制）。

粘合剂

这种绑定使用的粘合剂。有关详细信息，请参阅 Classpath 上的 Multiple Binders。

默认值：null（默认的 binder 将被使用，如果存在）。

消费者物业

以下绑定属性仅适用于输入绑定，并且必须以 `spring.cloud.stream.bindings.<channelName>.consumer.` 为前缀，例如 `spring.cloud.stream.bindings.input.consumer.concurrency=3`。

默认值可以使用前缀 `spring.cloud.stream.default.consumer` 设置，例如 `spring.cloud.stream.default.consumer.headerMode=raw`。

并发

入站消费者的并发性。

默认值：`1`。

分区

消费者是否从分区生产者接收数据。

默认值：`false`。

headerMode

设置为 `raw` 时，禁用输入头文件解析。仅适用于不支持消息头的消息中间件，并且需要头部嵌入。入站数据来自外部 Spring Cloud Stream 应用程序时很有用。

默认值：`embeddedHeaders`。

maxAttempts

如果处理失败，则尝试处理消息的次数（包括第一个）。设置为 1 以禁用重试。

默认值：`3`。

backOffInitialInterval

退避初始间隔重试。

默认值：`1000`。

backOffMaxInterval

最大回退间隔。

默认值：`10000`。

backOffMultiplier

退避倍数。

默认值：`2.0`。

instanceIndex

当设置为大于等于零的值时，允许自定义此消费者的实例索引（如果与 `spring.cloud.stream.instanceIndex` 不同）。设置为负值时，它将默认为 `spring.cloud.stream.instanceIndex`。

默认值：`-1`。

instanceCount

当设置为大于等于零的值时，允许自定义此消费者的实例计数（如果与 `spring.cloud.stream.instanceCount` 不同）。当设置为负值时，它将默认为 `spring.cloud.stream.instanceCount`。

默认值：`-1`。

制作人 Properties

以下绑定属性仅可用于输出绑定，并且必须以 `spring.cloud.stream.bindings.<channelName>.producer.` 为前缀，例如 `spring.cloud.stream.bindings.input.producer.partitionKeyExpression=payload.id`。

默认值可以使用前缀 `spring.cloud.stream.default.producer` 设置，例如 `spring.cloud.stream.default.producer.partitionKeyExpression=payload.id`。

partitionKeyExpression

一个确定如何分配出站数据的 SpEL 表达式。如果设置，或者如果设置了 `partitionKeyExtractorClass`，则该通道上的出站数据将被分区，并且 `partitionCount` 必须设置为大于 1 的值才能生效。这两个选项是相互排斥的。请参阅分区支持。

默认值：null。

partitionKeyExtractorClass

一个 `PartitionKeyExtractorStrategy` 实现。如果设置，或者如果设置了 `partitionKeyExpression`，则该通道上的出站数据将被分区，并且 `partitionCount` 必须设置为大于 1 的值才能生效。这两个选项是相互排斥的。请参阅分区支持。

默认值：null。

partitionSelectorClass

一个 `PartitionSelectorStrategy` 实现。与 `partitionSelectorExpression` 相互排斥。如果没有设置，则分区将被选为 `hashCode(key) % partitionCount`，其中 `key` 通过 `partitionKeyExpression` 或 `partitionKeyExtractorClass` 计算。

默认值：null。

partitionSelectorExpression

用于自定义分区选择的 SpEL 表达式。与 `partitionSelectorClass` 相互排斥。如果没有设置，则分区将被选为 `hashCode(key) % partitionCount`，其中 `key` 通过 `partitionKeyExpression` 或 `partitionKeyExtractorClass` 计算。

默认值：null。

partitionCount

如果启用分区，则数据的目标分区数。如果生产者被分区，则必须设置为大于 1 的值。在 Kafka，解释为提示; 而是使用更大的和目标主题的分区计数。

默认值：`1`。

requiredGroups

生成者必须确保消息传递的组合的逗号分隔列表，即使它们在创建之后启动（例如，通过在 RabbitMQ 中预先创建持久队列）。

headerMode

设置为 `raw` 时，禁用输出上的标题嵌入。仅适用于不支持消息头的消息中间件，并且需要头部嵌入。生成非 Spring Cloud Stream 应用程序的数据时很有用。

默认值：`embeddedHeaders`。

useNativeEncoding

当设置为`true`时，出站消息由客户端库直接序列化，必须相应配置（例如设置适当的 Kafka 生产者值序列化程序）。当使用此配置时，出站消息编组不是基于绑定的`contentType`。当使用本地编码时，消费者有责任使用适当的解码器（例如：Kafka 消费者价值解串器）来对入站消息进行反序列化。此外，当使用本机编码/解码时，`headerMode`属性将被忽略，标题不会嵌入到消息中。

默认值：`false`。

使用动态绑定目的地

除了通过`@EnableBinding`定义的通道之外，Spring Cloud Stream 允许应用程序将消息发送到动态绑定的目的地。这是有用的，例如，当目标目标需要在运行时确定。应用程序可以使用`@EnableBinding`注册自动注册的`BinderAwareChannelResolver`bean。

属性“spring.cloud.stream.dynamicDestinations”可用于将动态目标名称限制为预先已知的集合（白名单）。如果属性未设置，任何目的地都可以动态绑定。

可以直接使用`BinderAwareChannelResolver`，如以下示例所示，其中 REST 控制器使用路径变量来确定目标通道。

```
@EnableBinding
@Controller
public class SourceWithDynamicDestination {

      @Autowired
      private BinderAwareChannelResolver resolver;
```

```
	@RequestMapping(path = "/{target}", method = POST, consumes = "*/*")
	@ResponseStatus(HttpStatus.ACCEPTED)
	public void handleRequest(@RequestBody String body, @PathVariable("target") target,
		@RequestHeader(HttpHeaders.CONTENT_TYPE) Object contentType) {
		sendMessage(body, target, contentType);
	}

	private void sendMessage(String body, String target, Object contentType) {

	resolver.resolveDestination(target).send(MessageBuilder.createMessage(body,
				new MessageHeaders(Collections.singletonMap(MessageHeaders.CONTENT_TYPE, contentType))));
	}
}
```

在默认端口 8080 上启动应用程序后，发送以下数据时：

```
curl -H "Content-Type: application/json" -X POST -d "customer-1" http://localhost:8080/customers

curl -H "Content-Type: application/json" -X POST -d "order-1" http://localhost:8080/orders
```

目的地的客户和"订单"是在经纪人中创建的（例如：在 Rabbit 的情况下进行交换，或者在 Kafka 的情况下为主题），其名称为"客户"和"订单"，数据被发布到适当的目的地。

`BinderAwareChannelResolver`是通用的 Spring Integration `DestinationResolver`，可以注入其他组件。例如，在使用基于传入 JSON 消息的 `target` 字段的 SpEL 表达式的路由器中。

```
@EnableBinding
@Controller
public class SourceWithDynamicDestination {

	@Autowired
	private BinderAwareChannelResolver resolver;


	@RequestMapping(path = "/", method = POST, consumes
= "application/json")
	@ResponseStatus(HttpStatus.ACCEPTED)
	public void handleRequest(@RequestBody String body,
@RequestHeader(HttpHeaders.CONTENT_TYPE) Object
contentType) {
		sendMessage(body, contentType);
	}

	private void sendMessage(Object body, Object
contentType) {

	routerChannel().send(MessageBuilder.createMessage(b
ody,
				new
MessageHeaders(Collections.singletonMap(MessageHeaders.CO
NTENT_TYPE, contentType))));
	}

	@Bean(name = "routerChannel")
	public MessageChannel routerChannel() {
		return new DirectChannel();
	}

	@Bean
	@ServiceActivator(inputChannel = "routerChannel")
	public ExpressionEvaluatingRouter router() {
	 ExpressionEvaluatingRouter router =
	    new ExpressionEvaluatingRouter(new
SpelExpressionParser().parseExpression("payload.target"))
;
		router.setDefaultOutputChannelName("default-
output");
		router.setChannelResolver(resolver);
		return router;
	}
```

```
}
```

内容类型和转换

要允许您传播关于已生成消息的内容类型的信息，默认情况下，Spring Cloud Stream 附加 `contentType` 标头到出站消息。对于不直接支持头文件的中间件，Spring Cloud Stream 提供了自己的自动将邮件包裹在自己的信封中的机制。对于支持头文件的中间件，Spring Cloud Stream 应用程序可以从非 Spring Cloud Stream 应用程序接收具有给定内容类型的消息。

Spring Cloud Stream 可以通过两种方式处理基于此信息的消息：

- 通过其入站和出站渠道的 `contentType` 设置
- 通过对`@StreamListener`注释的方法执行的参数映射

Spring Cloud Stream 允许您使用绑定的 `spring.cloud.stream.bindings.<channelName>.content-type` 属性声明性地配置输入和输出的类型转换。请注意，一般类型转换也可以通过在应用程序中使用变压器轻松实现。目前，Spring Cloud Stream 本机支持流中常用的以下类型转换：

- 来自/从 **POJO 的 JSON**
- **JSON** /从 org.springframework.tuple.Tuple

- **对象**到/来自 **byte []**：用于远程传输的原始字节序列化，应用程序发出的字节，或使用 Java 序列化转换为字节（要求对象为 Serializable）
- **字符串**到/来自 **byte []**
- **对象到纯文本**（调用对象的 *toString* *()* 方法）

其中 *JSON* 表示包含 *JSON* 的字节数组或字符串有效负载。目前，对象可以从 JSON 字节数组或字符串转换。转换为 JSON 总是产生一个 String。

如果在出站通道上没有设置 `content-type` 属性，则 Spring Cloud Stream 将使用基于 Kryo 序列化框架的序列化程序对有效负载进行序列化。在目的地反序列化消息需要在接收者的类路径上存在有效载荷类。

MIME 类型

`content-type` 值被解析为媒体类型，例如 `application/json` 或 `text/plain;charset=UTF-8`。MIME 类型对于指示如何转换为 String 或 byte []内容特别有用。Spring Cloud Stream 还使用 MIME 类型格式来表示 Java 类型，使用具有 `type` 参数的一般类型 `application/x-java-object`。例如，`application/x-java-object;type=java.util.Map` 或 `application/x-java-object;type=com.bar.Foo` 可以设置为输入绑定的 `content-type` 属性。此外，Spring Cloud Stream 提供自定义 MIME 类型，特别是 `application/x-spring-tuple` 来指定元组。

MIME 类型和 Java 类型

类型转换 Spring Cloud Stream 提供的开箱即用如下表所示：“源有效载荷”是指转换前的有效载荷，“目标有效载荷”是指转换后的“有效载荷”。类型转换可以在“生产者”一侧（输出）或“消费者”一侧（输入）上进行。

来源有效载荷	目标有效载荷	`content-type` 标题（来源讯息）	`content-type` 标题（ 后）
POJO	JSON String	ignored	application/json
Tuple	JSON String	ignored	application/json
POJO	String (toString())	ignored	text/plain, java.lang.Stri
POJO	byte[] (java.io serialized)	ignored	application/x-java-serialized-object
JSON byte[] or String	POJO	application/json (or none)	application/x-java-obje
byte[] or String	Serializable	application/x-java-serialized-object	application/x-java-obje
JSON byte[] or String	Tuple	application/json (or none)	application/x-spring-tu
byte[]	String	any	text/plain, java.lang.Stri
String	byte[]	any	application/octet-strean

注意	转换适用于需要类型转换的有效内容。例如，如果应用程序生成带有 outputType = 有效载荷将不会从 XML 转换为 JSON。这是因为发送到出站通道的有效载荷已经是 换。同样重要的是要注意，当使用默认的序列化机制时，必须在发送和接收应用程 内容兼容。当应用程序代码在两个应用程序中独立更改时，这可能会产生问题，因 容。

提示 | 虽然入站和出站渠道都支持转换，但特别推荐将其用于转发出站邮件。对于入站邮件 `@StreamListener` 支持将自动执行转换。

自定义邮件转换

除了支持开箱即用的转换，Spring Cloud Stream 还支持注册您自己的邮件转换实现。这允许您以各种自定义格式（包括二进制）发送和接收数据，并将其与特定的 `contentTypes` 关联。Spring Cloud Stream 将所有类型为 `org.springframework.messaging.converter.MessageConverter` 的 bean 注册为自定义消息转换器以及开箱即用消息转换器。

如果您的消息转换器需要使用特定的 `content-type` 和目标类（用于输入和输出），则消息转换器需要扩展 `org.springframework.messaging.converter.AbstractMessageConverter`。对于使用`@StreamListener` 的转换，实现 `org.springframework.messaging.converter.MessageConverter` 的消息转换器就足够了。

以下是在 Spring Cloud Stream 应用程序中创建消息转换器 bean（内容类型为 `application/bar`）的示例:

```
@EnableBinding(Sink.class)
@SpringBootApplication
public static class SinkApplication {

  ...

  @Bean
  public MessageConverter customMessageConverter() {
    return new MyCustomMessageConverter();
```

```
  }
public class MyCustomMessageConverter extends
AbstractMessageConverter {

      public MyCustomMessageConverter() {
            super(new MimeType("application", "bar"));
      }

      @Override
  protected boolean supports(Class<?> clazz) {
    return (Bar.class == clazz);
  }

      @Override
      protected Object convertFromInternal(Message<?>
message, Class<?> targetClass, Object conversionHint) {
            Object payload = message.getPayload();
            return (payload instanceof Bar ? payload :
new Bar((byte[]) payload));
      }
}
```

Spring Cloud Stream 还为基于 Avro 的转换器和模式演进提供支持。详情请参阅具体章节。

`@StreamListener`和讯息转换

`@StreamListener`注释提供了一种方便的方式来转换传入的消息，而不需要指定输入通道的内容类型。在使用`@StreamListener`注释的方法的调度过程中，如果参数需要转换，将自动应用转换。

例如，让我们考虑一个带有`{"greeting":"Hello, world"}`的 String 内容的消息，并且在输入通道上收到`application/json`的`application/json`标题。让我们考虑接收它的以下应用程序：

```
public class GreetingMessage {

  String greeting;

  public String getGreeting() {
    return greeting;
  }

  public void setGreeting(String greeting) {
    this.greeting = greeting;
  }
}

@EnableBinding(Sink.class)
@EnableAutoConfiguration
public static class GreetingSink {

               @StreamListener(Sink.INPUT)
               public void receive(Greeting greeting) {
                       // handle Greeting
               }
       }
```

该方法的参数将自动填充包含 JSON 字符串的未编组形式的 POJO。

Schema 进化支持

Spring Cloud Stream 通过其 `spring-cloud-stream-schema` 模块为基于模式的消息转换器提供支持。目前，基于模式的消息转换器开箱即用的唯一序列化格式是 Apache Avro，在将来的版本中可以添加更多的格式。

Apache Avro 讯息转换器

`spring-cloud-stream-schema` 模块包含可用于 Apache Avro 序列化的两种类型的消息转换器：

- 使用序列化/反序列化对象的类信息的转换器，或者启动时已知位置的模式;
- 转换器使用模式注册表 - 他们在运行时定位模式，以及随着域对象的发展动态注册新模式。

具有模式支持的转换器

`AvroSchemaMessageConverter` 支持使用预定义模式或使用类中可用的模式信息（反射或包含在 `SpecificRecord`）中的序列化和反序列化消息。如果转换的目标类型是 `GenericRecord`，则必须设置模式。

对于使用它，您可以简单地将其添加到应用程序上下文中，可选地指定一个或多个 `MimeTypes` 将其关联。默认 `MimeType` 为 `application/avro`。

以下是在注册 Apache Avro `MessageConverter` 的宿应用程序中进行配置的示例，而不需要预定义的模式：

```
@EnableBinding(Sink.class)
@SpringBootApplication
public static class SinkApplication {

  ...

  @Bean
  public MessageConverter userMessageConverter() {
      return new
AvroSchemaMessageConverter(MimeType.valueOf("avro/bytes")
);
  }
```

```
}
```

相反，这里是一个应用程序，注册一个具有预定义模式的转换器，可以在类路径中找到：

```
@EnableBinding(Sink.class)
@SpringBootApplication
public static class SinkApplication {

  ...

  @Bean
  public MessageConverter userMessageConverter() {
      AvroSchemaMessageConverter converter = new
AvroSchemaMessageConverter(MimeType.valueOf("avro/bytes")
);
      converter.setSchemaLocation(new
ClassPathResource("schemas/User.avro"));
      return converter;
  }
}
```

为了了解模式注册表客户端转换器，我们将首先描述模式注册表支持。

Schema 注册表支持

大多数序列化模型，特别是旨在跨不同平台和语言进行可移植性的序列化模型，依赖于描述数据如何在二进制有效载荷中被序列化的模式。为了序列化数据然后解释它，发送方和接收方都必须访问描述二进制格式的模式。在某些情况下，可以从序列化的有效载荷类型或从反序列化时的目标类型中推断出模式，但是在许多情况下，应用程序可以从访问描述二进制数据格式的显式模式中受益。模式注册表允许您以文本格式（通常为 JSON）存储模式信息，并使该信息可访问需要

它的各种应用程序以二进制格式接收和发送数据。一个模式可以作为一个元组引用，它由

- 作为模式的逻辑名称的*主题*；
- 模式*版本*；
- 描述*数据*的二进制格式的模式*格式*。

Schema 注册服务器

Spring Cloud Stream 提供了模式注册表服务器实现。为了使用它，您可以简单地将 `spring-cloud-stream-schema-server` 工件添加到项目中，并使用 `@EnableSchemaRegistryServer` 注释，将模式注册表服务器 REST 控制器添加到应用程序中。此注释旨在与 Spring Boot Web 应用程序一起使用，服务器的监听端口由 `server.port` 设置控制。

`spring.cloud.stream.schema.server.path` 设置可用于控制模式服务器的根路径（特别是嵌入其他应用程序时）。

`spring.cloud.stream.schema.server.allowSchemaDeletion` 布尔设置可以删除模式。默认情况下，这是禁用的。

模式注册表服务器使用关系数据库来存储模式。默认情况下，它使用一个嵌入式数据库。您可以使用 Spring Boot SQL 数据库和 JDBC 配置选项自定义模式存储。

启用模式注册表的 Spring Boot 应用程序如下所示：

```
@SpringBootApplication
```

```
@EnableSchemaRegistryServer
public class SchemaRegistryServerApplication {
	public static void main(String[] args) {

	SpringApplication.run(SchemaRegistryServerApplicati
on.class, args);
	}
}
```

Schema 注册服务器 API

Schema 注册服务器 API 由以下操作组成：

`POST /`

注册一个新的架构

接受具有以下字段的 JSON 有效载荷：

- `subject` 模式主题;
- `format` 模式格式;
- `definition` 模式定义。

响应是 JSON 格式的模式对象，包含以下字段：

- `id` 模式标识;
- `subject` 模式主题;
- `format` 模式格式;
- `version` 模式版本;
- `definition` 模式定义。

`GET /{subject}/{format}/{version}`

根据其主题，格式和版本检索现有模式。

响应是 JSON 格式的模式对象，包含以下字段：

- `id` 模式标识;
- `subject` 模式主题;
- `format` 模式格式;
- `version` 模式版本;
- `definition` 模式定义。

`GET /{subject}/{format}`

根据其主题和格式检索现有模式的列表。

响应是 JSON 格式的每个模式对象的模式列表，包含以下字段：

- `id` 模式标识;
- `subject` 模式主题;
- `format` 模式格式;
- `version` 模式版本;
- `definition` 模式定义。

`GET /schemas/{id}`

通过其 id 来检索现有的模式。

响应是 JSON 格式的模式对象，包含以下字段：

- `id` 模式标识;
- `subject` 模式主题;
- `format` 模式格式;
- `version` 模式版本;
- `definition` 模式定义。

`DELETE /{subject}/{format}/{version}`

按其主题，格式和版本删除现有模式。

`DELETE /schemas/{id}`

按其 ID 删除现有模式。

`DELETE /{subject}`

按其主题删除现有模式。

注意	本说明仅适用于 Spring Cloud Stream 1.1.0.RELEASE 的用户。Spring Cloud Stream 1 `Schema` 对象，这是一些数据库实现中的关键字。为了避免将来发生任何冲突，从 1 的名称 `SCHEMA_REPOSITORY`。建议任何正在升级的 Spring Cloud Stream 1.1.0.REL 移到新表。

Schema 注册表客户端

与模式注册表服务器交互的客户端抽象是 `SchemaRegistryClient` 接口，具有

以下结构：

```
public interface SchemaRegistryClient {

	SchemaRegistrationResponse register(String subject,
String format, String schema);

	String fetch(SchemaReference schemaReference);

	String fetch(Integer id);

}
```

Spring Cloud Stream 提供了开箱即用的实现，用于与其自己的模式服务器交互，以及与 Confluent Schema 注册表进行交互。

可以使用`@EnableSchemaRegistryClient`配置 Spring Cloud Stream 模式注册表的客户端，如下所示：

```
@EnableBinding(Sink.class)
@SpringBootApplication
@EnableSchemaRegistryClient
public static class AvroSinkApplication {
  ...
}
```

注意	优化了默认转换器，以缓存来自远程服务器的模式，而且还会非常昂贵的 `parse()` 缓存响应的 `DefaultSchemaRegistryClient`。如果您打算直接在代码上使用客 响应的 bean。为此，只需将属性 `spring.cloud.stream.schemaRegistryCli` 性中即可。

Schema 注册表客户端属性

Schema 注册表客户端支持以下属性：

spring.cloud.stream.schemaRegistryClient.endpoint

模式服务器的位置。在设置时使用完整的 URL，包括协议（`http` 或 `https`），端口和上下文路径。

默认

http://localhost:8990/

spring.cloud.stream.schemaRegistryClient.cached

客户端是否应缓存模式服务器响应。通常设置为 `false`，因为缓存发生在消息转换器中。使用模式注册表客户端的客户端应将其设置为 `true`。

默认

`true`

Avro Schema 注册表客户端消息转换器

对于在应用程序上下文中注册了 `SchemaRegistryClient` bean 的 Spring Boot 应用程序，Spring Cloud Stream 将自动配置使用模式注册表客户端进行模式管理的 Apache Avro 消息转换器。这简化了模式演进，因为接收消息的应用程序可以轻松访问可与自己的读取器模式进行协调的写入器模式。

对于出站邮件，如果频道的内容类型设置为 `application/*+avro`，`MessageConverter` 将被激活，例如：

```
spring.cloud.stream.bindings.output.contentType=application/*+avro
```

在出站转换期间，消息转换器将尝试基于其类型推断出站消息的模式，并使用 `SchemaRegistryClient` 根据有效载荷类型将其注册到主题。如果已经找到相同的模式，那么将会检索对它的引用。如果没有，则将注册模式并提供新的版本号。该消息将使用 `application/[prefix].[subject].v[version]+avro`

的方案 `contentType` 头发送，其中 `prefix` 是可配置的，并且从有效载荷类型推导出 `subject`。

例如，类型为 `User` 的消息可以作为内容类型为 `application/vnd.user.v2+avro` 的二进制有效载荷发送，其中 `user` 是主题，`2` 是版本号。

当接收到消息时，转换器将从传入消息的头部推断出模式引用，并尝试检索它。该模式将在反序列化过程中用作写入器模式。

Avro Schema 注册表消息转换器属性

如果您已通过设置 `spring.cloud.stream.bindings.output.contentType=application/*+avro` 启用基于 Avro 的模式注册表客户端，则可以使用以下属性自定义注册的行为。

spring.cloud.stream.schema.avro.dynamicSchemaGenerationEnabled

如果您希望转换器使用反射从 POJO 推断 Schema，则启用。

默认

`false`

spring.cloud.stream.schema.avro.readerSchema

Avro 通过查看编写器模式（源有效载荷）和读取器模式（应用程序有效负载）来比较模式版本，查看 Avro 文档以获取更多信息。如果设置，这将覆盖模式服务器上的任何查找，并将本地模式用作读取器模式。

默认

`null`

spring.cloud.stream.schema.avro.schemaLocations

使用 Schema 服务器注册此属性中列出的任何`.avsc`文件。

默认

`empty`

spring.cloud.stream.schema.avro.prefix

要在 Content-Type 头上使用的前缀。

默认

`vnd`

Schema 注册和解决

为了更好地了解 Spring Cloud Stream 注册和解决新模式以及其使用 Avro 模式比较功能，我们将提供两个单独的子部分：一个用于注册，一个用于解析模式。

Schema 注册流程（序列化）

注册过程的第一部分是从通过信道发送的有效载荷中提取模式。Avro 类型，如`SpecificRecord`或`GenericRecord`已经包含一个模式，可以从实例中立即检索。在 POJO 的情况下，如果属性`spring.cloud.stream.schema.avro.dynamicSchemaGenerationEnabled`设置为`true`（默认），则会推断出一个模式。

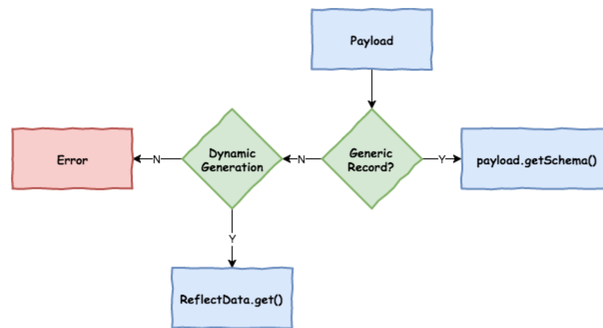


图 10. Schema Writer Resolution Process

一旦获得了架构，转换器就会从远程服务器加载其元数据（版本）。首先，它查询本地缓存，如果没有找到它，则将数据提交到将使用版本控制信息回复的服务器。转换器将始终缓存结果，以避免为每个需要序列化的新消息查询 Schema 服务器的开销。

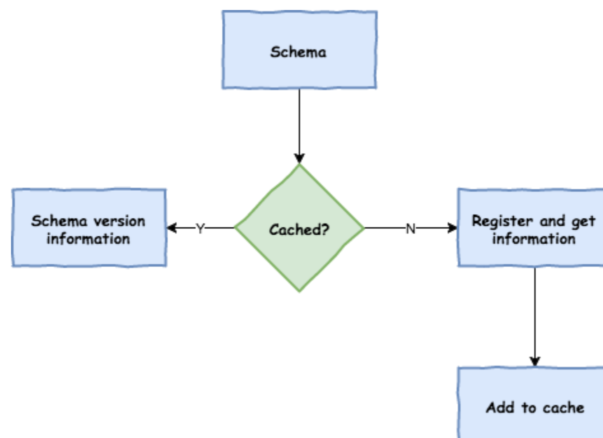


图 11. Schema 注册流程

使用模式版本信息，转换器设置消息的 `contentType` 头，以携带版本信息，如 `application/vnd.user.v1+avro`

Schema 解析过程（反序列化）

当读取包含版本信息的消息（即，具有上述方案的 `contentType` 标头）时，转换器将查询 Schema 服务器以获取消息的**写入器**架构。一旦找到传入消息的正确

架构，它就会检索读取器架构，并使用 Avro 的架构解析支持将其读入读取器定义（设置默认值和缺少的属性）。

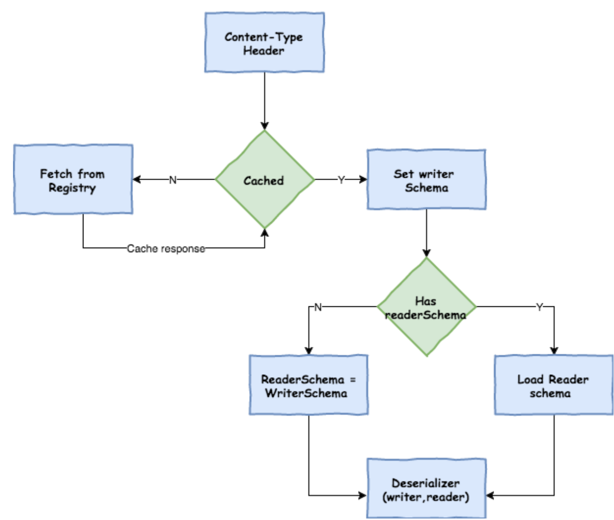


图 12. Schema 阅读决议程序

注意	了解编写器架构（写入消息的应用程序）和读取器架构（接收应用程序）之间的区 了解此过程。Spring Cloud Stream 将始终提取 writer 模式以确定如何读取消息。如 您需要确保为您的应用程序正确设置了 readerSchema。

应用间通信

连接多个应用程序实例

虽然 Spring Cloud Stream 使个人 Spring Boot 应用程序轻松连接到消息传递系统，但是 Spring Cloud Stream 的典型场景是创建多应用程序管道，其中微服务应用程序将数据发送给彼此。您可以通过将相邻应用程序的输入和输出目标相关联来实现此场景。

假设设计要求时间源应用程序将数据发送到日志接收应用程序，则可以在两个应用程序中使用名为 `ticktock` 的公共目标进行绑定。

时间来源（具有频道名称 `output`）将设置以下属性：

```
spring.cloud.stream.bindings.output.destination=ticktock
```

日志接收器（通道名称为 `input`）将设置以下属性：

```
spring.cloud.stream.bindings.input.destination=ticktock
```

实例索引和实例计数

当扩展 Spring Cloud Stream 应用程序时，每个实例都可以接收有关同一个应用程序的其他实例数量以及自己的实例索引的信息。Spring Cloud Stream 通过 `spring.cloud.stream.instanceCount` 和 `spring.cloud.stream.instanceIndex` 属性执行此操作。例如，如果 HDFS 宿应用程序有三个实例，则所有三个实例将 `spring.cloud.stream.instanceCount` 设置为 `3`，并且各个应用程序将 `spring.cloud.stream.instanceIndex` 设置为 `0`，`1` 和 `2`。

当通过 Spring Cloud 数据流部署 Spring Cloud Stream 应用程序时，这些属性将自动配置；当 Spring Cloud Stream 应用程序独立启动时，必须正确设置这些属性。默认情况下，`spring.cloud.stream.instanceCount` 为 `1`，`spring.cloud.stream.instanceIndex` 为 `0`。

在放大的情况下，这两个属性的正确配置对于解决分区行为（见下文）一般很重要，并且某些绑定器（例如，Kafka binder）总是需要这两个属性，以确保该数据在多个消费者实例之间正确分割。

分区

配置输出绑定进行分区

输出绑定被配置为通过设置其唯一的一个 `partitionKeyExpression` 或 `partitionKeyExtractorClass` 属性以及其 `partitionCount` 属性来发送分区数据。例如，以下是一个有效和典型的配置：

```
spring.cloud.stream.bindings.output.producer.partitionKey
Expression=payload.id
spring.cloud.stream.bindings.output.producer.partitionCou
nt=5
```

基于上述示例配置，使用以下逻辑将数据发送到目标分区。

基于 `partitionKeyExpression`，为发送到分区输出通道的每个消息计算分区密钥的值。`partitionKeyExpression` 是一个 Spel 表达式，它根据出站消息进行评估，以提取分区键。

如果 SpEL 表达式不足以满足您的需要，您可以通过将属性 `partitionKeyExtractorClass` 设置为实现 `org.springframework.cloud.stream.binder.PartitionKeyExtractorStrategy` 接口的类来计算分区键值。虽然 Spel 表达式通常足够，但更复杂的

情况可能会使用自定义实现策略。在这种情况下，属性“partitionKeyExtractorClass”可以设置如下：

```
spring.cloud.stream.bindings.output.producer.partitionKeyExtractorClass=com.example.MyKeyExtractor
spring.cloud.stream.bindings.output.producer.partitionCount=5
```

一旦计算了消息密钥，分区选择过程将确定目标分区为 `0` 和 `partitionCount - 1` 之间的值。在大多数情况下，默认计算基于公式 `key.hashCode() % partitionCount`。这可以通过设置要针对'key'（通过 `partitionSelectorExpression` 属性）进行评估的 Spel 表达式或通过设置 `org.springframework.cloud.stream.binder.PartitionSelectorStrategy` 实现（通过 `partitionSelectorClass` 属性））进行自定义。

“partitionSelectorExpression”和“partitionSelectorClass”的绑定级属性可以类似于上述示例中指定的“partitionKeyExpression”和“partitionKeyExtractorClass”属性的类型。可以为更高级的场景配置其他属性，如以下部分所述。

Spring - 管理的自定义 `PartitionKeyExtractorClass` 实现

在上面的示例中，`MyKeyExtractor` 之类的自定义策略由 Spring Cloud Stream 直接实例化。在某些情况下，必须将这样的自定义策略实现创建为 Spring bean，以便能够由 Spring 管理，以便它可以执行依赖注入，属性绑定等。可以通过将其配置为应用程序上下文中的@Bean，并使用完全限定类名作为 bean 的名称，如以下示例所示。

```
@Bean(name="com.example.MyKeyExtractor")
public MyKeyExtractor extractor() {
```

```
    return new MyKeyExtractor();
}
```

作为 Spring bean，自定义策略从 Spring bean 的完整生命周期中受益。例如，如果实现需要直接访问应用程序上下文，则可以实现“ApplicationContextAware”。

配置输入绑定进行分区

输入绑定（通道名称为 `input`）被配置为通过在应用程序本身设置其 `partitioned` 属性以及 `instanceIndex` 和 `instanceCount` 属性来接收分区数据，如以下示例：

```
spring.cloud.stream.bindings.input.consumer.partitioned=true
spring.cloud.stream.instanceIndex=3
spring.cloud.stream.instanceCount=5
```

`instanceCount` 值表示数据需要分区的应用程序实例的总数，`instanceIndex` 必须是 `0` 和 `instanceCount - 1` 之间的多个实例的唯一值。实例索引帮助每个应用程序实例识别从其接收数据的唯一分区（或者在 Kafka 的分区集合的情况下）。重要的是正确设置两个值，以确保所有数据都被使用，并且应用程序实例接收到互斥数据集。

虽然使用多个实例进行分区数据处理的场景可能会在独立情况下进行复杂化，但是通过将输入和输出值正确填充并依赖于运行时基础架构，Spring Cloud 数据流可以显着简化流程。提供有关实例索引和实例计数的信息。

测试

Spring Cloud Stream 支持测试您的微服务应用程序，而无需连接到消息系统。您可以使用 `spring-cloud-stream-test-support` 库提供的 `TestSupportBinder`，可以将其作为测试依赖项添加到应用程序中：

```
  <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-stream-test-
support</artifactId>
      <scope>test</scope>
  </dependency>
```

注意	`TestSupportBinder` 使用 Spring Boot 自动配置机制取代类路径中找到的其他绑定 时，请确保正在使用 `test` 范围。

`TestSupportBinder` 允许用户与绑定的频道进行交互，并检查应用程序发送和接收的消息

对于出站消息通道，`TestSupportBinder` 注册单个订户，并将应用程序发送的消息保留在 `MessageCollector` 中。它们可以在测试过程中被检索，并对它们做出断言。

用户还可以将消息发送到入站消息通道，以便消费者应用程序可以使用消息。以下示例显示了如何在处理器上测试输入和输出通道。

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=
SpringBootTest.WebEnvironment.RANDOM_PORT)
public class ExampleTest {

  @Autowired
  private Processor processor;

  @Autowired
  private MessageCollector messageCollector;
```

```
  @Test
  @SuppressWarnings("unchecked")
  public void testWiring() {
    Message<String> message = new
GenericMessage<>("hello");
    processor.input().send(message);
    Message<String> received = (Message<String>)
messageCollector.forChannel(processor.output()).poll();
    assertThat(received.getPayload(), equalTo("hello
world"));
  }


  @SpringBootApplication
  @EnableBinding(Processor.class)
  public static class MyProcessor {

    @Autowired
    private Processor channels;

    @Transformer(inputChannel = Processor.INPUT,
outputChannel = Processor.OUTPUT)
    public String transform(String in) {
      return in + " world";
    }
  }
}
```

在上面的示例中，我们正在创建一个具有输入和输出通道的应用程序，通过 `Processor` 接口绑定。绑定的接口被注入测试，所以我们可以访问这两个通道。我们正在输入频道发送消息，我们使用 Spring Cloud Stream 测试支持提供的 `MessageCollector` 来捕获消息已经被发送到输出通道。收到消息后，我们可以验证组件是否正常工作。

健康指标

Spring Cloud Stream 为粘合剂提供健康指标。它以 `binders` 的名义注册，可以通过设置 `management.health.binders.enabled` 属性启用或禁用。

指标发射器

Spring Cloud Stream 提供了一个名为 `spring-cloud-stream-metrics` 的模块，可以用来从 Spring Boot 度量端点到命名通道发出任何可用度量。该模块允许运营商从流应用收集指标，而不依赖轮询其端点。

当您设置度量绑定的目标名称（例如 `spring.cloud.stream.bindings.applicationMetrics.destination=<DESTINATION_NAME>`）时，该模块将被激活。可以以与任何其他生成器绑定相似的方式配置 `applicationMetrics`。`applicationMetrics` 的 `contentType` 默认设置为 `application/json`。

以下属性可用于自定义度量标准的排放：

spring.cloud.stream.metrics.key

要发射的度量的名称。应该是每个应用程序的唯一值。

默认

`${spring.application.name:${vcap.application.name:${spring.config.name:application}}}`

spring.cloud.stream.metrics.prefix

前缀字符串，以前缀到度量键。

默认值：``

spring.cloud.stream.metrics.properties

就像 `includes` 选项一样，它允许将白名单应用程序属性添加到度量有效负载

默认值：null。

有关度量导出过程的详细概述，请参见 Spring Boot 参考文档。Spring Cloud Stream 提供了一个名为 `application` 的指标导出器，可以通过常规 Spring Boot 指标配置属性进行配置。

可以通过使用出口商的全局 Spring Boot 配置设置或使用特定于导出器的属性来配置导出器。要使用全局配置设置，属性应以 `spring.metric.export` 为前缀（例如 `spring.metric.export.includes=integration**`）。这些配置选项将适用于所有出口商（除非它们的配置不同）。或者，如果要使用与其他出口商不同的配置设置（例如，限制发布的度量数量），则可以使用前缀 `spring.metrics.export.triggers.application` 配置 Spring Cloud Stream 提供的度量导出器（例如 `spring.metrics.export.triggers.application.includes=integration**`）。

注意	由于 Spring Boot 的轻松约束，所包含的属性的值可能与原始值稍有不同。 作为经验法则，度量导出器将尝试使用点符号（例如 `JAVA_HOME` 成为 `java.home` 规范化的目标是使下游用户能够始终如一地接收属性名称，无论它们如何设置在受 `spring.application.name` 或 `SPRING_APPLICATION_NAME` 始终会生成 `spri`

以下是通过以下命令以 JSON 格式发布到频道的数据的示例:

```
java -jar time-source.jar \
    --
spring.cloud.stream.bindings.applicationMetrics.destinati
on=someMetrics \
    --
spring.cloud.stream.metrics.properties=spring.application
** \
    --
spring.metrics.export.includes=integration.channel.input*
*,integration.channel.output**
```

得到的 JSON 是:


```
{
  "name":"time-source",
  "metrics":[
     {

"name":"integration.channel.output.errorRate.mean",
        "value":0.0,
        "timestamp":"2017-04-11T16:56:35.790Z"
     },
     {
        "name":"integration.channel.output.errorRate.max",
        "value":0.0,
        "timestamp":"2017-04-11T16:56:35.790Z"
     },
     {
        "name":"integration.channel.output.errorRate.min",
        "value":0.0,
        "timestamp":"2017-04-11T16:56:35.790Z"
     },
     {

"name":"integration.channel.output.errorRate.stdev",
        "value":0.0,
        "timestamp":"2017-04-11T16:56:35.790Z"
     },
     {

"name":"integration.channel.output.errorRate.count",
```

```
        "value":0.0,
        "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
        "name":"integration.channel.output.sendCount",
        "value":6.0,
        "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
        "name":"integration.channel.output.sendRate.mean",
        "value":0.994885872292989,
        "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
        "name":"integration.channel.output.sendRate.max",
        "value":1.006247080013156,
        "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {
        "name":"integration.channel.output.sendRate.min",
        "value":1.0012035220116378,
        "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {

"name":"integration.channel.output.sendRate.stdev",
        "value":6.505181111084848E-4,
        "timestamp":"2017-04-11T16:56:35.790Z"
      },
      {

"name":"integration.channel.output.sendRate.count",
        "value":6.0,
        "timestamp":"2017-04-11T16:56:35.790Z"
      }
   ],
   "createdTime":"2017-04-11T20:56:35.790Z",
   "properties":{
     "spring.application.name":"time-source",
     "spring.application.index":"0"
   }
}
```

样品

对于 Spring Cloud Stream 示例，请参阅 GitHub 上的 spring-cloud-stream 样本存储库。

入门

要开始创建 Spring Cloud Stream 应用程序，请访问 Spring Initializr 并创建一个名为“GreetingSource”的新 Maven 项目。在下拉菜单中选择 Spring Boot {supported-spring-boot-version}。在“ *搜索依赖关系*”文本框中键入 `Stream Rabbit` 或 `Stream Kafka`，具体取决于您要使用的 binder。

接下来，在与 `GreetingSourceApplication` 类相同的包中创建一个新类 `GreetingSource`。给它以下代码：

```
import
org.springframework.cloud.stream.annotation.EnableBinding
;
import org.springframework.cloud.stream.messaging.Source;
import
org.springframework.integration.annotation.InboundChannel
Adapter;

@EnableBinding(Source.class)
public class GreetingSource {

    @InboundChannelAdapter(Source.OUTPUT)
    public String greet() {
        return "hello world " + System.currentTimeMillis();
    }
}
```

`@EnableBinding`注释是触发 Spring Integration 基础架构组件的创建。具体来说，它将创建一个 Kafka 连接工厂，一个 Kafka 出站通道适配器，并在 Source 界面中定义消息通道：

```
public interface Source {

  String OUTPUT = "output";

  @Output(Source.OUTPUT)
  MessageChannel output();

}
```

自动配置还创建一个默认轮询器，以便每秒调用`greet()`方法一次。标准的 Spring Integration `@InboundChannelAdapter`注释使用返回值作为消息的有效内容向源的输出通道发送消息。

要测试驱动此设置，请运行 Kafka 消息代理。一个简单的方法是使用 Docker 镜像：

```
# On OS X
$ docker run -p 2181:2181 -p 9092:9092 --env
ADVERTISED_HOST=`docker-machine ip \`docker-machine
active\`` --env ADVERTISED_PORT=9092 spotify/kafka

# On Linux
$ docker run -p 2181:2181 -p 9092:9092 --env
ADVERTISED_HOST=localhost --env ADVERTISED_PORT=9092
spotify/kafka
```

构建应用程序：

```
./mvnw clean package
```

消费者应用程序以类似的方式进行编码。返回 Initializr 并创建另一个名为 LoggingSink 的项目。然后在与类 `LoggingSinkApplication` 相同的包中创建一个新类 `LoggingSink`，并使用以下代码：

```
import
org.springframework.cloud.stream.annotation.EnableBinding
;
import
org.springframework.cloud.stream.annotation.StreamListene
r;
import org.springframework.cloud.stream.messaging.Sink;

@EnableBinding(Sink.class)
public class LoggingSink {

    @StreamListener(Sink.INPUT)
    public void log(String message) {
        System.out.println(message);
    }
}
```

构建应用程序：

```
./mvnw clean package
```

要将 GreetingSource 应用程序连接到 LoggingSink 应用程序，每个应用程序必须共享相同的目标名称。启动这两个应用程序如下所示，您将看到消费者应用程序打印"hello world"和时间戳到控制台：

```
cd GreetingSource
java -jar target/GreetingSource-0.0.1-SNAPSHOT.jar --
spring.cloud.stream.bindings.output.destination=mydest

cd LoggingSink
java -jar target/LoggingSink-0.0.1-SNAPSHOT.jar --
server.port=8090 --
spring.cloud.stream.bindings.input.destination=mydest
```

（不同的服务器端口可以防止两个应用程序中用于维护 Spring Boot 执行器端点的 HTTP 端口的冲突。）

LoggingSink 应用程序的输出将如下所示：

```
[           main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started
on port(s): 8090 (http)
[           main] com.example.LoggingSinkApplication      :
Started LoggingSinkApplication in 6.828 seconds (JVM
running for 7.371)
hello world 1458595076731
hello world 1458595077732
hello world 1458595078733
hello world 1458595079734
hello world 1458595080735
```

Binder 实施

Apache Kafka Binder

用法

对于使用 Apache Kafka 绑定器，您只需要使用以下 Maven 坐标将其添加到您的 Spring Cloud Stream 应用程序：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-
kafka</artifactId>
</dependency>
```

或者，您也可以使用 Spring Cloud Stream Kafka Starter。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-
kafka</artifactId>
</dependency>
```

Apache Kafka Binder 概述

以下可以看到 Apache Kafka 绑定器操作的简化图。

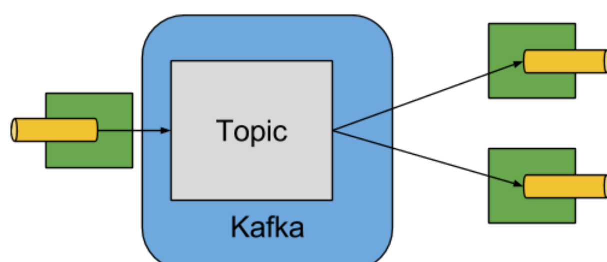


图 13. Kafka Binder

Apache Kafka Binder 实现将每个目标映射到 Apache Kafka 主题。消费者组织直接映射到相同的 Apache Kafka 概念。分区也直接映射到 Apache Kafka 分区。

配置选项

本节包含 Apache Kafka 绑定器使用的配置选项。

有关 binder 的常见配置选项和属性，请参阅核心文档。

Kafka Binder Properties

spring.cloud.stream.kafka.binder.brokers

Kafka 活页夹将连接的经纪人列表。

默认值：`localhost`。

spring.cloud.stream.kafka.binder.defaultBrokerPort

`brokers` 允许使用或不使用端口信息指定的主机（例如，`host1,host2:port2`）。当在代理列表中没有配置端口时，这将设置默认端口。

默认值：`9092`。

spring.cloud.stream.kafka.binder.zkNodes

Kafka 绑定器可以连接的 ZooKeeper 节点列表。

默认值：`localhost`。

spring.cloud.stream.kafka.binder.defaultZkPort

`zkNodes` 允许使用或不使用端口信息指定的主机（例如，`host1,host2:port2`）。当在节点列表中没有配置端口时，这将设置默认端口。

默认值：`2181`。

spring.cloud.stream.kafka.binder.configuration

客户端属性（生产者和消费者）的密钥/值映射传递给由绑定器创建的所有客户端。由于这些属性将被生产者和消费者使用，所以使用应该限于常见的属性，特别是安全设置。

默认值：空地图。

spring.cloud.stream.kafka.binder.headers

将由活页夹传送的自定义标题列表。

默认值：空。

spring.cloud.stream.kafka.binder.offsetUpdateTimeWindow

以毫秒为单位的频率（以毫秒为单位）保存偏移量。`0` 忽略。

默认值：`10000`。

spring.cloud.stream.kafka.binder.offsetUpdateCount

频率，更新次数，哪些消耗的偏移量会持续存在。`0` 忽略。与 `offsetUpdateTimeWindow` 相互排斥。

默认值：`0`。

spring.cloud.stream.kafka.binder.requiredAcks

经纪人所需的 acks 数量。

默认值：`1`。

spring.cloud.stream.kafka.binder.minPartitionCount

只有设置 `autoCreateTopics` 或 `autoAddPartitions` 才有效。绑定器在其生成/消耗数据的主题上配置的全局最小分区数。它可以由生产者的 `partitionCount` 设置或生产者的 `instanceCount` * `concurrency` 设置的值替代（如果更大）。

默认值：`1`。

spring.cloud.stream.kafka.binder.replicationFactor

如果`autoCreateTopics`处于活动状态，则自动创建主题的复制因子。

默认值：`1`。

spring.cloud.stream.kafka.binder.autoCreateTopics

如果设置为`true`，绑定器将自动创建新主题。如果设置为`false`，则绑定器将依赖于已配置的主题。在后一种情况下，如果主题不存在，则绑定器将无法启动。值得注意的是，此设置与代理的`auto.topic.create.enable`设置无关，并不影响它：如果服务器设置为自动创建主题，则可以将其创建为元数据检索请求的一部分，并使用默认代理设置。

默认值：`true`。

spring.cloud.stream.kafka.binder.autoAddPartitions

如果设置为`true`，则绑定器将根据需要创建新的分区。如果设置为`false`，则绑定器将依赖于已配置的主题的分区大小。如果目标主题的分区计数小于预期值，则绑定器将无法启动。

默认值：`false`。

spring.cloud.stream.kafka.binder.socketBufferSize

Kafka 消费者使用的套接字缓冲区的大小（以字节为单位）。

默认值：`2097152`。

Kafka 消费者 Properties

以下属性仅适用于 Kafka 消费者，必须以 `spring.cloud.stream.kafka.bindings.<channelName>.consumer.` 为前缀。

autoRebalanceEnabled

当 `true`，主题分区将在消费者组的成员之间自动重新平衡。当 `false` 根据 `spring.cloud.stream.instanceCount` 和 `spring.cloud.stream.instanceIndex` 为每个消费者分配一组固定的分区。这需要在每个启动的实例上适当地设置 `spring.cloud.stream.instanceCount` 和 `spring.cloud.stream.instanceIndex` 属性。在这种情况下，属性 `spring.cloud.stream.instanceCount` 通常必须大于 1。

默认值：`true`。

autoCommitOffset

是否在处理邮件时自动提交偏移量。如果设置为 `false`，则入站消息中将显示带有

`org.springframework.kafka.support.Acknowledgment`类型的密钥`kafka_acknowledgment`的报头。应用程序可以使用此标头来确认消息。有关详细信息，请参阅示例部分。当此属性设置为`false`时，Kafka binder 将 ack 模式设置为`org.springframework.kafka.listener.AbstractMessageListenerContainer.AckMode.MANUAL`。

默认值：`true`。

autoCommitOnError

只有`autoCommitOffset`设置为`true`才有效。如果设置为`false`，它会禁止导致错误的邮件的自动提交，并且只会为成功的邮件执行提交，允许流在上次成功处理的邮件中自动重播，以防持续发生故障。如果设置为`true`，它将始终自动提交（如果启用了自动提交）。如果没有设置（默认），它实际上具有与`enableDlq`相同的值，如果它们被发送到 DLQ，则自动提交错误的消息，否则不提交它们。

默认值：未设置。

recoveryInterval

连接恢复尝试之间的间隔，以毫秒为单位。

默认值：`5000`。

resetOffsets

是否将消费者的偏移量重置为`startOffset`提供的值。

默认值：`false`。

开始偏移

新组的起始偏移量，或`resetOffsets`为`true`时的起始偏移量。允许的值：`earliest`，`latest`。如果消费者组被明确设置为消费者'绑定'（通过`spring.cloud.stream.bindings.<channelName>.group`），那么'startOffset'设置为`earliest`; 否则对于`anonymous`消费者组，设置为`latest`。

默认值：null（相当于`earliest`）。

enableDlq

当设置为 true 时，它将为消费者发送启用 DLQ 行为。默认情况下，导致错误的邮件将转发到名为`error.<destination>.<group>`的主题。DLQ 主题名称可以通过属性`dlqName`配置。对于错误数量相对较少并且重播整个原始主题可能太麻烦的情况，这为更常见的 Kafka 重播场景提供了另一种选择。

默认值：`false`。

组态

使用包含通用 Kafka 消费者属性的键/值对映射。

默认值：空地图。

dlqName

接收错误消息的 DLQ 主题的名称。

默认值：null（如果未指定，将导致错误的消息将转发到名为 `error.<destination>.<group>`的主题）。

Kafka 生产者 Properties

以下属性仅适用于 Kafka 生产者，必须以 `spring.cloud.stream.kafka.bindings.<channelName>.producer.`为前缀。

缓冲区大小

上限（以字节为单位），Kafka 生产者将在发送之前尝试批量的数据量。

默认值：`16384`。

同步

生产者是否是同步的

默认值：`false`。

batchTimeout

生产者在发送之前等待多长时间，以便允许更多消息在同一批次中累积。（通常，生产者根本不等待，并且简单地发送在先前发送进行中累积的所有消息。）非零值可能会以延迟为代价增加吞吐量。

默认值：`0`。

组态

使用包含通用 Kafka 生产者属性的键/值对映射。

默认值：空地图。

注意	Kafka 绑定器将使用生产者的 `partitionCount` 设置作为提示，以创建具有给定分 `minPartitionCount` 一起使用，最多两个为正在使用的值） 。配置绑定器的 mi `partitionCount` 时要小心，因为将使用较大的值。如果一个主题已经存在较小的 被禁用（默认值），则绑定器将无法启动。如果一个主题已经存在较小的分区计数， 则会添加新的分区。如果一个主题已经存在的分区数量大于（`minPartitionCoun` 将使用现有的分区计数。

用法示例

在本节中，我们举例说明了上述属性在具体情况下的使用。

示例：设置 `autoCommitOffset` false 并依赖手动确认。

该示例说明了如何在消费者应用程序中手动确认偏移量。

此示例要求 `spring.cloud.stream.kafka.bindings.input.consumer.autoCommitOffset` 设置为 false。使用相应的输入通道名称作为示例。

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class ManuallyAcknowdledgingConsumer {

 public static void main(String[] args) {
```

```
SpringApplication.run(ManuallyAcknowdledgingConsumer.clas
s, args);
 }

 @StreamListener(Sink.INPUT)
 public void process(Message<?> message) {
     Acknowledgment acknowledgment =
message.getHeaders().get(KafkaHeaders.ACKNOWLEDGMENT,
Acknowledgment.class);
     if (acknowledgment != null) {
         System.out.println("Acknowledgment provided");
         acknowledgment.acknowledge();
     }
 }
}
```

示例：安全配置

Apache Kafka 0.9 支持客户端和代理商之间的安全连接。要充分利用此功能，请遵循汇编文档中的 Apache Kafka 文档以及 Kafka 0.9 安全性指导原则。使用 `spring.cloud.stream.kafka.binder.configuration` 选项为绑定器创建的所有客户端设置安全属性。

例如，要将 `security.protocol` 设置为 `SASL_SSL`，请设置：

```
spring.cloud.stream.kafka.binder.configuration.security.p
rotocol=SASL_SSL
```

所有其他安全属性可以以类似的方式设置。

使用 Kerberos 时，请按照参考文档中的说明创建和引用 JAAS 配置。

Spring Cloud Stream 支持使用 JAAS 配置文件并使用 Spring Boot 属性将 JAAS 配置信息传递到应用程序。

使用 JAAS 配置文件

可以通过使用系统属性为 Spring Cloud Stream 应用程序设置 JAAS 和（可选）krb5 文件位置。以下是使用 JAAS 配置文件启动带有 SASL 和 Kerberos 的 Spring Cloud Stream 应用程序的示例：

```
 java -Djava.security.auth.login.config=/path.to/kafka_client_jaas.conf -jar log.jar \
   --spring.cloud.stream.kafka.binder.brokers=secure.server:9092 \
   --spring.cloud.stream.kafka.binder.zkNodes=secure.zookeeper:2181 \
   --spring.cloud.stream.bindings.input.destination=stream.ticktock \
   --spring.cloud.stream.kafka.binder.configuration.security.protocol=SASL_PLAINTEXT
```

使用 Spring Boot 属性

作为使用 JAAS 配置文件的替代方案，Spring Cloud Stream 提供了一种使用 Spring Boot 属性为 Spring Cloud Stream 应用程序设置 JAAS 配置的机制。

以下属性可用于配置 Kafka 客户端的登录上下文。

spring.cloud.stream.kafka.binder.jaas.loginModule

　　登录模块名称。在正常情况下不需要设置。

　　默认值：`com.sun.security.auth.module.Krb5LoginModule`。

spring.cloud.stream.kafka.binder.jaas.controlFlag

登录模块的控制标志。

默认值：`required`。

spring.cloud.stream.kafka.binder.jaas.options

使用包含登录模块选项的键/值对映射。

默认值：空地图。

以下是使用 Spring Boot 配置属性启动带有 SASL 和 Kerberos 的 Spring Cloud Stream 应用程序的示例：

```
 java --
spring.cloud.stream.kafka.binder.brokers=secure.server:90
92 \
  --
spring.cloud.stream.kafka.binder.zkNodes=secure.zookeeper
:2181 \
  --
spring.cloud.stream.bindings.input.destination=stream.tic
ktock \
  --
spring.cloud.stream.kafka.binder.autoCreateTopics=false \
  --
spring.cloud.stream.kafka.binder.configuration.security.p
rotocol=SASL_PLAINTEXT \
  --
spring.cloud.stream.kafka.binder.jaas.options.useKeyTab=t
rue \
  --
spring.cloud.stream.kafka.binder.jaas.options.storeKey=tr
ue \
  --
spring.cloud.stream.kafka.binder.jaas.options.keyTab=/etc
/security/keytabs/kafka_client.keytab \
```

```
  --
spring.cloud.stream.kafka.binder.jaas.options.principal=k
afka-client-1@EXAMPLE.COM
```

这相当于以下 JAAS 文件:

```
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/etc/security/keytabs/kafka_client.keytab"
    principal="kafka-client-1@EXAMPLE.COM";
};
```

如果所需的主题已经存在于代理上，或将由管理员创建，则自动创建可以被关闭，并且仅需要发送客户端 JAAS 属性。作为设置 `spring.cloud.stream.kafka.binder.autoCreateTopics` 的替代方法，您可以简单地从应用程序中删除代理依赖关系。有关详细信息，请参阅基于绑定器的应用程序的类路径中排除 Kafka 代理 jar。

注意	不要在同一应用程序中混合 JAAS 配置文件和 Spring Boot 属性。如果`-Djava.secu` 已存在，则 Spring Cloud Stream 将忽略 Spring Boot 属性。
注意	使用 `autoCreateTopics` 和 `autoAddPartitions` 如果使用 Kerberos，请务必小 Zookeeper 中没有管理权限的主体，并且依赖 Spring Cloud Stream 创建/修改主题可 议您使用 Kafka 工具管理性地创建主题并管理 ACL。

使用绑定器与 Apache Kafka 0.10

Spring Cloud Stream Kafka binder 中的默认 Kafka 支持是针对 Kafka 版本 0.10.1.1 的。粘合剂还支持连接到其他 0.10 版本和 0.9 客户端。为了做到这一点，当你创建包含你的应用程序的项目时，包括 `spring-cloud-starter-stream-`

kafka，你通常会对默认的绑定器做。然后将这些依赖项添加到 pom.xml 文件中的<dependencies>部分的顶部以覆盖依赖关系。

以下是将应用程序降级到 0.10.0.1 的示例。由于它仍在 0.10 行，因此可以保留默认的 spring-kafka 和 spring-integration-kafka 版本。

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka_2.11</artifactId>
  <version>0.10.0.1</version>
  <exclusions>
    <exclusion>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>0.10.0.1</version>
</dependency>
```

这是使用 0.9.0.1 版本的另一个例子。

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
  <version>1.0.5.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-kafka</artifactId>
  <version>2.0.1.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka_2.11</artifactId>
  <version>0.9.0.1</version>
  <exclusions>
```

```
    <exclusion>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>0.9.0.1</version>
</dependency>
```

注意	以上版本仅为了举例而提供。为获得最佳效果，我们建议您使用最新的 0.10 兼容版

从基于绑定器的应用程序的类路径中排除 Kafka 代理 jar

Apache Kafka Binder 使用作为 Apache Kafka 服务器库一部分的管理实用程序来创建和重新配置主题。如果在运行时不需要包含 Apache Kafka 服务器库及其依赖关系，因为应用程序将依赖于管理中配置的主题，Kafka binder 允许排除 Apache Kafka 服务器依赖关系从应用程序。

如果您使用上述建议的 Kafka 依赖关系的非默认版本，则只需要包含 kafka 代理依赖项。如果您使用默认的 Kafka 版本，请确保从 `spring-cloud-starter-stream-kafka` 依赖关系中排除 kafka broker jar，如下所示。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-
kafka</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka_2.11</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

如果您排除 Apache Kafka 服务器依赖关系，并且该主题不在服务器上，那么如果在服务器上启用了自动主题创建，则 Apache Kafka 代理将创建该主题。请注意，如果您依赖此，则 Kafka 服务器将使用默认数量的分区和复制因子。另一方面，如果在服务器上禁用自动主题创建，则在运行应用程序之前必须注意创建具有所需数量分区的主题。

如果要完全控制分区的分配方式，请保留默认设置，即不要排除 kafka 代理程序 jar，并确保将 `spring.cloud.stream.kafka.binder.autoCreateTopics` 设置为 `true`，这是默认设置。

Dead-Letter 主题处理

因为不可能预料到用户如何处理死信消息，所以框架不提供任何标准的机制来处理它们。如果死刑的原因是短暂的，您可能希望将邮件路由到原始主题。但是，如果问题是一个永久性的问题，那可能会导致无限循环。以下 `spring-boot` 应用程序是如何将这些消息路由到原始主题的示例，但在三次尝试后将其移动到第三个“停车场”主题。该应用程序只是从死信主题中读取的另一个 spring-cloud-stream 应用程序。5 秒内没有收到消息时终止。

这些示例假定原始目的地是 `so8400out`，而消费者组是 `so8400`。

有几个注意事项

- 当主应用程序未运行时，请考虑仅运行重新路由。否则，瞬态错误的重试将很快用尽。

- 或者，使用两阶段方法 - 使用此应用程序路由到第三个主题，另一个则从那里路由到主题。
- 由于这种技术使用消息标头来跟踪重试，所以它不会与 `headerMode=raw` 一起使用。在这种情况下，请考虑将一些数据添加到有效载荷（主应用程序可以忽略）。
- 必须将 `x-retries` 添加到 `headers` 属性 `spring.cloud.stream.kafka.binder.headers=x-retries` 和主应用程序，以便标头在应用程序之间传输。
- 由于 kafka 是发布/订阅，所以重播的消息将被发送给每个消费者组，即使是那些首次成功处理消息的消费者组。

application.properties

```
spring.cloud.stream.bindings.input.group=so8400replay
spring.cloud.stream.bindings.input.destination=error.so8400out.so8400

spring.cloud.stream.bindings.output.destination=so8400out
spring.cloud.stream.bindings.output.producer.partitioned=true

spring.cloud.stream.bindings.parkingLot.destination=so8400in.parkingLot
spring.cloud.stream.bindings.parkingLot.producer.partitioned=true

spring.cloud.stream.kafka.binder.configuration.auto.offset.reset=earliest

spring.cloud.stream.kafka.binder.headers=x-retries
```

应用

```
@SpringBootApplication
@EnableBinding(TwoOutputProcessor.class)
```

```
public class ReRouteDlqKApplication implements
CommandLineRunner {

   private static final String X_RETRIES_HEADER = "x-
retries";

   public static void main(String[] args) {
       SpringApplication.run(ReRouteDlqKApplication.class,
args).close();
   }

   private final AtomicInteger processed = new
AtomicInteger();

   @Autowired
   private MessageChannel parkingLot;

   @StreamListener(Processor.INPUT)
   @SendTo(Processor.OUTPUT)
   public Message<?> reRoute(Message<?> failed) {
       processed.incrementAndGet();
       Integer retries =
failed.getHeaders().get(X_RETRIES_HEADER, Integer.class);
       if (retries == null) {
           System.out.println("First retry for " +
failed);
           return MessageBuilder.fromMessage(failed)
                   .setHeader(X_RETRIES_HEADER, new
Integer(1))
                   .setHeader(BinderHeaders.PARTITION_OVERRI
DE,

failed.getHeaders().get(KafkaHeaders.RECEIVED_PARTITION_I
D))
                   .build();
       }
       else if (retries.intValue() < 3) {
           System.out.println("Another retry for " +
failed);
           return MessageBuilder.fromMessage(failed)
                   .setHeader(X_RETRIES_HEADER, new
Integer(retries.intValue() + 1))
                   .setHeader(BinderHeaders.PARTITION_OVERRI
DE,
```

```
failed.getHeaders().get(KafkaHeaders.RECEIVED_PARTITION_I
D))
                    .build();
        }
        else {
            System.out.println("Retries exhausted for " +
failed);

parkingLot.send(MessageBuilder.fromMessage(failed)
                    .setHeader(BinderHeaders.PARTITION_OVERRI
DE,

failed.getHeaders().get(KafkaHeaders.RECEIVED_PARTITION_I
D))
                    .build());
        }
        return null;
    }

    @Override
    public void run(String... args) throws Exception {
        while (true) {
            int count = this.processed.get();
            Thread.sleep(5000);
            if (count == this.processed.get()) {
                System.out.println("Idle, terminating");
                return;
            }
        }
    }

    public interface TwoOutputProcessor extends Processor
{

        @Output("parkingLot")
        MessageChannel parkingLot();

    }

}
```

RabbitMQ Binder

用法

对于使用 RabbitMQ 绑定器，您只需要使用以下 Maven 坐标将其添加到您的 Spring Cloud Stream 应用程序：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-
rabbit</artifactId>
</dependency>
```

或者，您也可以使用 Spring Cloud Stream RabbitMQ 入门。

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-
rabbit</artifactId>
</dependency>
```

RabbitMQ Binder 概述

以下可以看到 RabbitMQ 活页夹的操作简化图。

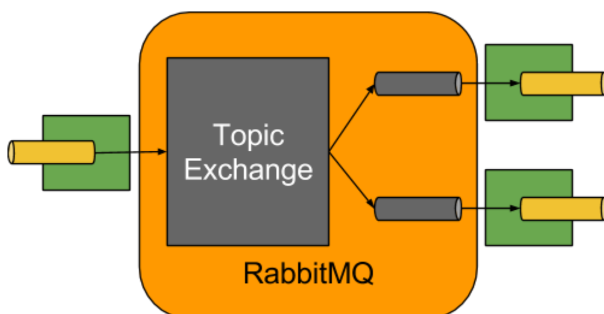


图 14. RabbitMQ Binder

RabbitMQ Binder 实现将每个目的地映射到 `TopicExchange`。对于每个消费者组，`Queue` 将绑定到该 `TopicExchange`。每个消费者实例对其组的 `Queue` 具

有相应的 RabbitMQ `Consumer` 实例。对于分区生成器/消费者，队列后缀为分区索引，并使用分区索引作为路由密钥。

使用 `autoBindDlq` 选项，您可以选择配置绑定器来创建和配置死信队列（DLQ）（以及死信交换 `DLX`）。死信队列具有目标名称，附有 `.dlq`。如果重试启用（`maxAttempts > 1`），则会将失败的消息传递到 DLQ。如果禁用重试（`maxAttempts = 1`），则应将 `requeueRejected` 设置为 `false`（默认），以使失败的消息将路由到 DLQ，而不是重新排队。此外，`republishToDlq` 导致绑定器向 DLQ 发布失败的消息（而不是拒绝它）；这使得能够将标题中的附加信息添加到消息中，例如 `x-exception-stacktrace` 头中的堆栈跟踪。此选项不需要重试启用; 一次尝试后，您可以重新发布失败的消息。从*版本 1.2* 开始，您可以配置重新发布的消息传递模式; 见财产 `republishDeliveryMode`。

重要	将 `requeueRejected` 设置为 `true` 将导致消息被重新排序并重新发送，这可能不 一般来说，最好通过将 `maxAttempts` 设置为大于 1，或将 `republishToDlq` 设置

有关这些属性的更多信息，请参阅 [RabbitMQ Binder Properties]。

框架不提供消耗死信消息（或重新路由到主队列）的任何标准机制。[Dead-Letter 队列处理]中描述了一些选项。

注意	在 Spring Cloud Stream 应用程序中使用**多个** RabbitMQ 绑定器时，禁用“RabbitAuto RabbitAutoConfiguration 应用于两个绑定器的相同配置很重要。

配置选项

本节包含特定于 RabbitMQ Binder 和绑定频道的设置。

有关通用绑定配置选项和属性，请参阅 Spring Cloud Stream 核心文档。

RabbitMQ Binder Properties

默认情况下，RabbitMQ binder 使用 Spring Boot 的 `ConnectionFactory`，因此它支持 RabbitMQ 的所有 Spring Boot 配置选项。（有关参考，请参阅 Spring Boot 文档。）RabbitMQ 配置选项使用 `spring.rabbitmq` 前缀。

除 Spring Boot 选项之外，RabbitMQ binder 还支持以下属性：

spring.cloud.stream.rabbit.binder.adminAddresses

RabbitMQ 管理插件网址的逗号分隔列表。仅在 `nodes` 包含多个条目时使用。此列表中的每个条目必须在 `spring.rabbitmq.addresses` 中具有相应的条目。

默认值：空。

spring.cloud.stream.rabbit.binder.nodes

RabbitMQ 节点名称的逗号分隔列表。当多个条目用于查找队列所在的服务器地址时。此列表中的每个条目必须在 `spring.rabbitmq.addresses` 中具有相应的条目。

默认值：空。

spring.cloud.stream.rabbit.binder.compressionLevel

压缩绑定的压缩级别。见 `java.util.zip.Deflater`。

默认值：`1`（BEST_LEVEL）。

RabbitMQ 消费者 Properties

以下属性仅适用于 Rabbit 消费者，并且必须以 `spring.cloud.stream.rabbit.bindings.<channelName>.consumer.` 为前缀。

acknowledgeMode

确认模式。

默认值：`AUTO`。

autoBindDlq

是否自动声明 DLQ 并将其绑定到绑定器 DLX。

默认值：`false`。

bindingRoutingKey

将队列绑定到交换机的路由密钥（如果 `bindQueue` 为 `true`）。将附加分区目的地`-<instanceIndex>`。

默认值：`#`。

bindQueue

是否将队列绑定到目的地交换机？如果您已经设置了自己的基础设施并且先前已经创建/绑定了队列，请设置为 `false`。

默认值：`true`。

deadLetterQueueName

DLQ 的名称

默认值：`prefix+destination.dlq`

deadLetterExchange

分配给队列的 DLX; 如果 autoBindDlq 为 true

默认值：'prefix + DLX'

deadLetterRoutingKey

一个死信路由密钥分配给队列; 如果 autoBindDlq 为 true

默认值：`destination`

declareExchange

是否为目的地申报交换。

默认值：`true`。

delayedExchange

是否将交换声明为 `Delayed Message Exchange` - 需要在代理上延迟的消息交换插件。`x-delayed-type` 参数设置为 `exchangeType`。

默认值：`false`。

dlqDeadLetterExchange

如果 DLQ 被声明，则将 DLX 分配给该队列

默认值：`none`

dlqDeadLetterRoutingKey

如果 DLQ 被声明，则会将一个死信路由密钥分配给该队列; 默认无

默认值：`none`

dlqExpires

未使用的死信队列被删除多久（ms）

默认值：`no expiration`

dlqMaxLength

死信队列中的最大消息数

默认值：`no limit`

dlqMaxLengthBytes

来自所有消息的死信队列中的最大字节数

默认值：`no limit`

dlqMaxPriority

死信队列中消息的最大优先级（0-255）

默认值：`none`

dlqTtl

声明（ms）时默认适用于死信队列的时间

默认值：`no limit`

durableSubscription

订阅是否应该耐用。仅当`group`也被设置时才有效。

默认值：`true`。

exchangeAutoDelete

如果`declareExchange`为真，则交换机是否应该自动删除（删除最后一个队列后删除）。

默认值：`true`。

exchangeDurable

如果`declareExchange`为真，则交换应该是否持久（经纪人重新启动）。

默认值：`true`。

exchangeType

交换类型; 非分区目的地的 `direct`， `fanout` 或 `topic` `direct` 或 `topic` 分区目的地。

默认值：`topic`。

到期

未使用的队列被删除多久（ms）

默认值：`no expiration`

headerPatterns

要从入站邮件映射的头文件。

默认值：`['*']`（所有标题）。

maxConcurrency

最大消费者人数

默认值：`1`。

最长长度

队列中最大消息数

默认值：`no limit`

maxLengthBytes

连接恢复尝试之间的间隔，以毫秒为单位。

来自所有消息的队列中最大字节数

默认：`no limit`

maxPriority

队列中消息的最大优先级（0-255）

默认

`none`

预取

预取计数。

默认值：`1`。

字首

要添加到`destination`和队列名称的前缀。

默认值：“”。

recoveryInterval

连接恢复尝试之间的间隔，以毫秒为单位。

默认值：`5000`。

requeueRejected

在重试禁用或重新发布 ToDlq 是否为 false 时，是否应重新发送传递失败。

默认值：`false`。

republishDeliveryMode

当 `republishToDlq` 为 `true` 时，指定重新发布的邮件的传递模式。

默认值：`DeliveryMode.PERSISTENT`

republishToDlq

默认情况下，尝试重试后失败的消息将被拒绝。如果配置了死信队列（DLQ），则 RabbitMQ 将将失败的消息（未更改）路由到 DLQ。如果设置为 `true`，则绑定器将重新发布具有附加头的 DLQ 的失败消息，包括最终失败的原因的异常消息和堆栈跟踪。

默认值：false

交易

是否使用交易渠道。

默认值：`false`。

TTL

声明（ms）时默认适用于队列的时间

默认值：`no limit`

txSize

阿克斯之间的交付次数。

默认值：`1`。

兔子生产者 Properties

以下属性仅适用于 Rabbit 生产者，必须以 `spring.cloud.stream.rabbit.bindings.<channelName>.producer.` 为前缀。

autoBindDlq

是否自动声明 DLQ 并将其绑定到绑定器 DLX。

默认值：`false`。

batchingEnabled

是否启用生产者的消息批处理。

默认值：`false`。

BATCHSIZE

批量启动时要缓冲的消息数。

默认值：`100`。

batchBufferLimit

默认值：`10000`。

batchTimeout

默认值：`5000`。

bindingRoutingKey

将队列绑定到交换机的路由密钥（如果`bindQueue`为`true`）。仅适用于非分区目的地。仅适用于`requiredGroups`，然后仅提供给这些组。

默认值：`#`。

bindQueue

是否将队列绑定到目的地交换机？如果您已经设置了自己的基础架构并且先前已经创建/绑定了队列，请设置为`false`。仅适用于`requiredGroups`，然后仅提供给这些组。

默认值：`true`。

压缩

发送时是否应压缩数据。

默认值：`false`。

deadLetterQueueName

DLQ 的名称仅适用于 `requiredGroups`，仅适用于这些组。

默认值：`prefix+destination.dlq`

deadLetterExchange

分配给队列的 DLX; 如果 autoBindDlq 为 true 只适用于 `requiredGroups`，然后只提供给这些组。

默认值：'prefix + DLX'

deadLetterRoutingKey

一个死信路由密钥分配给队列; 如果 autoBindDlq 为 true 只适用于 `requiredGroups`，然后只提供给这些组。

默认值：`destination`

declareExchange

是否为目的地申报交换。

默认值：`true`。

延迟

评估应用于消息（`x-delay`头）的延迟的 Spel 表达式 - 如果交换不是延迟的消息交换，则不起作用。

默认值：No `x-delay`头设置。

delayedExchange

是否将交换声明为`Delayed Message Exchange` - 需要经纪人上的延迟消息交换插件。`x-delayed-type`参数设置为`exchangeType`。

默认值：`false`。

deliveryMode

交货方式。

默认值：`PERSISTENT`。

dlqDeadLetterExchange

如果 DLQ 被声明，则分配给该队列的 DLX 只适用于`requiredGroups`，然后仅提供给这些组。

默认值：`none`

dlqDeadLetterRoutingKey

如果 DLQ 被声明，则会将一个死信路由密钥分配给该队列; 默认值 none 仅在提供`requiredGroups`时才适用，然后仅适用于这些组。

默认值：`none`

dlqExpires

未使用的死信队列被删除之前多久（ms）仅适用于 `requiredGroups`，然后仅提供给这些组。

默认值：`no expiration`

dlqMaxLength

死信队列中的最大消息数仅适用于 `requiredGroups`，仅适用于这些组。

默认值：`no limit`

dlqMaxLengthBytes

来自所有消息的死信队列中的最大字节数仅适用于 `requiredGroups`，然后仅提供给这些组。

默认值：`no limit`

dlqMaxPriority

死信队列中消息的最大优先级（0-255）仅适用于 `requiredGroups`，然后仅提供给这些组。

默认值：`none`

dlqTtl

声明（ms）的默认时间适用于死信队列仅适用于 `requiredGroups`，然后仅提供给这些组。

默认值：`no limit`

exchangeAutoDelete

如果 `declareExchange` 为真，则交换机是否应该自动删除（删除最后一个队列后删除）。

默认值：`true`。

exchangeDurable

如果 `declareExchange` 为真，则交换应该是持久的（经纪人重新启动）。

默认值：`true`。

exchangeType

交换类型; `direct`，`fanout` 或 `topic`; `direct` 或 `topic`。

默认值：`topic`。

到期

在未使用的队列被删除之前多久（ms）仅适用于 `requiredGroups`，然后只提供给这些组。

默认值：`no expiration`

headerPatterns

要将标头映射到出站邮件的模式。

默认值：`['*']`（所有标题）。

最长长度

队列中最大消息数仅适用于 `requiredGroups`，仅适用于这些组。

默认值：`no limit`

maxLengthBytes

来自所有消息的队列中最大字节数仅适用于 `requiredGroups`，仅适用于这些组。

默认值：`no limit`

maxPriority

队列中消息的最大优先级（0-255）仅适用于 `requiredGroups`，仅适用于这些组。

默认

`none`

字首

要添加到`destination`交换机名称的前缀。

默认值：“”。

routingKeyExpression

一个 SpEL 表达式来确定在发布消息时使用的路由密钥。

默认值：`destination`或`destination-<partition>`分区目的地。

交易

是否使用交易渠道。

默认值：`false`。

T
T
L

声明时默认适用于队列的时间（ms）仅适用于 `requiredGroups`，然后仅适用于这些组。

默认值：`no limit`

注意	在 RabbitMQ 的情况下，内容类型头可以由外部应用程序设置。Spring Cloud Stream 括通常不支持头文件的 Kafka）的传输的扩展内部协议的一部分）。

重试 RabbitMQ Binder

概观

在绑定器中启用重试时，侦听器容器线程将被挂起，以配置任何后退时段。在单个消费者需要严格排序时，这可能很重要，但是对于其他用例，它可以防止在该线程上处理其他消息。使用绑定器重试的另一种方法是设置死机字符随着时间生活在死信队列（DLQ）上，以及 DLQ 本身的死信配置。有关这里讨论的属性的更多信息，请参阅 RabbitMQ Binder Properties。启用此功能的示例配置：

- 将 `autoBindDlq` 设置为 `true` - 绑定器将创建一个 DLQ; 您可以选择在 `deadLetterQueueName` 中指定一个名称
- 将 `dlqTtl` 设置为您要在重新投递之间等待的退出时间

- 将 `dlqDeadLetterExchange` 设置为默认交换 - DLQ 的过期消息将被路由到原始队列，因为默认 `deadLetterRoutingKey` 是队列名称（`destination.group`）

要强制一个消息被填字，抛出一个 `AmqpRejectAndDontRequeueException`，或设置 `requeueRejected` 到 `true` 并抛出任何异常。

循环将继续没有结束，这对于短暂的问题是很好的，但是您可能想在一些尝试后放弃。幸运的是，RabbitMQ 提供了 `x-death` 标题，允许您确定发生了多少个周期。

在放弃之后确认一则消息，抛出一个 `ImmediateAcknowledgeAmqpException`。

把它放在一起

```
---
spring.cloud.stream.bindings.input.destination=myDestinat
ion
spring.cloud.stream.bindings.input.group=consumerGroup
#disable binder retries
spring.cloud.stream.bindings.input.consumer.max-
attempts=1
#dlx/dlq setup
spring.cloud.stream.rabbit.bindings.input.consumer.auto-
bind-dlq=true
spring.cloud.stream.rabbit.bindings.input.consumer.dlq-
ttl=5000
spring.cloud.stream.rabbit.bindings.input.consumer.dlq-
dead-letter-exchange=
---
```

此配置创建一个与通配符路由密钥`#`交换主题的队列 `myDestination.consumerGroup` 的交换 `myDestination`。它创建一个绑定到具有路由密钥 `myDestination.consumerGroup` 的直接交换 `DLX` 的 DLQ。当消息被拒绝时，它们被路由到 DLQ。5 秒钟后，消息过期，并使用队列名称作为路由密钥路由到原始队列。

Spring Boot 申请

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class XDeathApplication {

    public static void main(String[] args) {
        SpringApplication.run(XDeathApplication.class,
args);
    }

    @StreamListener(Sink.INPUT)
    public void listen(String in, @Header(name = "x-
death", required = false) Map<?,?> death) {
        if (death != null && death.get("count").equals(3L))
{
            // giving up - don't send to DLX
            throw new
ImmediateAcknowledgeAmqpException("Failed after 4
attempts");
        }
        throw new
AmqpRejectAndDontRequeueException("failed");
    }

}
```

请注意，`x-death` 标题中的 count 属性是 `Long`。

Dead-Letter 队列处理

因为不可能预料到用户如何处理死信消息，所以框架不提供任何标准的机制来处理它们。如果死刑的原因是暂时的，您可能希望将邮件路由到原始队列。但是，如果问题是一个永久性的问题，那可能会导致无限循环。以下 `spring-boot` 应用程序是如何将这些消息路由到原始队列的示例，但是在三次尝试之后将其移动到第三个“停车场”队列。第二个例子使用 RabbitMQ 延迟消息交换来向被重新排序的消息引入延迟。在这个例子中，每次尝试的延迟都会增加。这些示例使用 `@RabbitListener` 从 DLQ 接收消息，您也可以在批处理过程中使用 `RabbitTemplate.receive()`。

这些示例假定原始目的地是 `so8400in`，消费者组是 `so8400`。

非分区目的地

前两个示例是目的地**未**分区。

```
@SpringBootApplication
public class ReRouteDlqApplication {

    private static final String ORIGINAL_QUEUE =
"so8400in.so8400";

    private static final String DLQ = ORIGINAL_QUEUE +
".dlq";

    private static final String PARKING_LOT =
ORIGINAL_QUEUE + ".parkingLot";

    private static final String X_RETRIES_HEADER = "x-
retries";

    public static void main(String[] args) throws
Exception {
        ConfigurableApplicationContext context =
SpringApplication.run(ReRouteDlqApplication.class, args);
```

```
        System.out.println("Hit enter to terminate");
        System.in.read();
        context.close();
    }

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @RabbitListener(queues = DLQ)
    public void rePublish(Message failedMessage) {
        Integer retriesHeader = (Integer)
failedMessage.getMessageProperties().getHeaders().get(X_R
ETRIES_HEADER);
        if (retriesHeader == null) {
            retriesHeader = Integer.valueOf(0);
        }
        if (retriesHeader < 3) {

failedMessage.getMessageProperties().getHeaders().put(X_R
ETRIES_HEADER, retriesHeader + 1);
            this.rabbitTemplate.send(ORIGINAL_QUEUE,
failedMessage);
        }
        else {
            this.rabbitTemplate.send(PARKING_LOT,
failedMessage);
        }
    }

    @Bean
    public Queue parkingLot() {
        return new Queue(PARKING_LOT);
    }

}
@SpringBootApplication
public class ReRouteDlqApplication {

    private static final String ORIGINAL_QUEUE =
"so8400in.so8400";

    private static final String DLQ = ORIGINAL_QUEUE +
".dlq";
```

```
    private static final String PARKING_LOT =
ORIGINAL_QUEUE + ".parkingLot";

    private static final String X_RETRIES_HEADER = "x-
retries";

    private static final String DELAY_EXCHANGE =
"dlqReRouter";

    public static void main(String[] args) throws
Exception {
        ConfigurableApplicationContext context =
SpringApplication.run(ReRouteDlqApplication.class, args);
        System.out.println("Hit enter to terminate");
        System.in.read();
        context.close();
    }

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @RabbitListener(queues = DLQ)
    public void rePublish(Message failedMessage) {
        Map<String, Object> headers =
failedMessage.getMessageProperties().getHeaders();
        Integer retriesHeader = (Integer)
headers.get(X_RETRIES_HEADER);
        if (retriesHeader == null) {
            retriesHeader = Integer.valueOf(0);
        }
        if (retriesHeader < 3) {
            headers.put(X_RETRIES_HEADER, retriesHeader +
1);
            headers.put("x-delay", 5000 * retriesHeader);
            this.rabbitTemplate.send(DELAY_EXCHANGE,
ORIGINAL_QUEUE, failedMessage);
        }
        else {
            this.rabbitTemplate.send(PARKING_LOT,
failedMessage);
        }
    }

    @Bean
```

```
    public DirectExchange delayExchange() {
        DirectExchange exchange = new
DirectExchange(DELAY_EXCHANGE);
        exchange.setDelayed(true);
        return exchange;
    }

    @Bean
    public Binding bindOriginalToDelay() {
        return BindingBuilder.bind(new
Queue(ORIGINAL_QUEUE)).to(delayExchange()).with(ORIGINAL_
QUEUE);
    }

    @Bean
    public Queue parkingLot() {
        return new Queue(PARKING_LOT);
    }

}
```

分区目的地

对于分区目的地，所有分区都有一个 DLQ，我们从头部确定原始队列。

republishToDlq = FALSE

当 `republishToDlq` 为 `false` 时，RabbitMQ 将消息发布到 DLX / DLQ，其中包含有关原始目的地信息的 `x-death` 标题。

```
@SpringBootApplication
public class ReRouteDlqApplication {

      private static final String ORIGINAL_QUEUE =
"so8400in.so8400";

      private static final String DLQ = ORIGINAL_QUEUE +
".dlq";

      private static final String PARKING_LOT =
ORIGINAL_QUEUE + ".parkingLot";
```

```
	private static final String X_DEATH_HEADER = "x-
death";

	private static final String X_RETRIES_HEADER = "x-
retries";

	public static void main(String[] args) throws
Exception {
		ConfigurableApplicationContext context =
SpringApplication.run(ReRouteDlqApplication.class, args);
		System.out.println("Hit enter to
terminate");
		System.in.read();
		context.close();
	}

	@Autowired
	private RabbitTemplate rabbitTemplate;

	@SuppressWarnings("unchecked")
	@RabbitListener(queues = DLQ)
	public void rePublish(Message failedMessage) {
		Map<String, Object> headers =
failedMessage.getMessageProperties().getHeaders();
		Integer retriesHeader = (Integer)
headers.get(X_RETRIES_HEADER);
		if (retriesHeader == null) {
			retriesHeader = Integer.valueOf(0);
		}
		if (retriesHeader < 3) {
			headers.put(X_RETRIES_HEADER,
retriesHeader + 1);
			List<Map<String, ?>> xDeath =
(List<Map<String, ?>>) headers.get(X_DEATH_HEADER);
			String exchange = (String)
xDeath.get(0).get("exchange");
			List<String> routingKeys =
(List<String>) xDeath.get(0).get("routing-keys");
			this.rabbitTemplate.send(exchange,
routingKeys.get(0), failedMessage);
		}
		else {
```

```
                    this.rabbitTemplate.send(PARKING_LOT,
failedMessage);
             }
      }

      @Bean
      public Queue parkingLot() {
             return new Queue(PARKING_LOT);
      }

}
```

republishToDlq =真

当 `republishToDlq` 为 `true` 时，重新发布恢复器将原始交换和路由密钥添加到标题。

```
@SpringBootApplication
public class ReRouteDlqApplication {

      private static final String ORIGINAL_QUEUE =
"so8400in.so8400";

      private static final String DLQ = ORIGINAL_QUEUE +
".dlq";

      private static final String PARKING_LOT =
ORIGINAL_QUEUE + ".parkingLot";

      private static final String X_RETRIES_HEADER = "x-
retries";

      private static final String
X_ORIGINAL_EXCHANGE_HEADER =
RepublishMessageRecoverer.X_ORIGINAL_EXCHANGE;

      private static final String
X_ORIGINAL_ROUTING_KEY_HEADER =
RepublishMessageRecoverer.X_ORIGINAL_ROUTING_KEY;

      public static void main(String[] args) throws
Exception {
```

```
		ConfigurableApplicationContext context =
SpringApplication.run(ReRouteDlqApplication.class, args);
		System.out.println("Hit enter to
terminate");
		System.in.read();
		context.close();
	}

	@Autowired
	private RabbitTemplate rabbitTemplate;

	@RabbitListener(queues = DLQ)
	public void rePublish(Message failedMessage) {
		Map<String, Object> headers =
failedMessage.getMessageProperties().getHeaders();
		Integer retriesHeader = (Integer)
headers.get(X_RETRIES_HEADER);
		if (retriesHeader == null) {
			retriesHeader = Integer.valueOf(0);
		}
		if (retriesHeader < 3) {
			headers.put(X_RETRIES_HEADER,
retriesHeader + 1);
			String exchange = (String)
headers.get(X_ORIGINAL_EXCHANGE_HEADER);
			String originalRoutingKey = (String)
headers.get(X_ORIGINAL_ROUTING_KEY_HEADER);
			this.rabbitTemplate.send(exchange,
originalRoutingKey, failedMessage);
		}
		else {
			this.rabbitTemplate.send(PARKING_LOT,
failedMessage);
		}
	}

	@Bean
	public Queue parkingLot() {
		return new Queue(PARKING_LOT);
	}

}
```

Spring Cloud Bus

Spring Cloud Bus 将分布式系统的节点与轻量级消息代理链接。这可以用于广播状态更改（例如配置更改）或其他管理指令。一个关键的想法是，总线就像一个分布式执行器，用于扩展的 Spring Boot 应用程序，但也可以用作应用程序之间的通信通道。目前唯一的实现是使用 AMQP 代理作为传输，但是相同的基本功能集（还有一些取决于传输）在其他传输的路线图上。

注意	Spring Cloud 根据非限制性 Apache 2.0 许可证发布。如果您想为文档的这一部分做 中找到项目中的源代码和问题跟踪器。

快速开始

Spring Cloud Bus 的工作原理是添加 Spring Boot 自动配置，如果它在类路径中检测到自身。所有您需要做的是启用总线是将 `spring-cloud-starter-bus-amqp` 或 `spring-cloud-starter-bus-kafka` 添加到您的依赖关系管理中，并且 Spring Cloud 负责其余部分。确保代理（RabbitMQ 或 Kafka）可用和配置：在本地主机上运行，您不应该做任何事情，但如果您远程运行使用 Spring Cloud 连接器或 Spring Boot 定义经纪人凭据的约定，例如 Rabbit

application.yml

```
spring:
  rabbitmq:
    host: mybroker.com
    port: 5672
    username: user
    password: secret
```

总线当前支持向所有节点发送消息，用于特定服务的所有节点（由 Eureka 定义）。未来可能会添加更多的选择器标准（即，仅数据中心 Y 中的服务 X 节点等）。`/bus/*`执行器命名空间下还有一些 http 端点。目前有两个实施。第一个 `/bus/env` 发送密钥/值对来更新每个节点的 Spring 环境。第二个，`/bus/refresh`，将重新加载每个应用程序的配置，就好像他们在他们的 `/refresh` 端点上都被 ping 过。

注意	总线起动器覆盖了 Rabbit 和 Kafka，因为这是两种最常用的实现方式，但是 Spring `spring-cloud-bus` 结合使用。

处理实例

HTTP 端点接受“目的地”参数，例如“/ bus / refresh？ destination = customers：9000”，其中目的地是 `ApplicationContext` ID。如果 ID 由总线上的一个实例拥有，那么它将处理消息，所有其他实例将忽略它。Spring Boot 将 `ContextIdApplicationContextInitializer` 中的 ID 设置为 `spring.application.name`，活动配置文件和 `server.port` 的组合。

寻址服务的所有实例

“destination”参数用于 Spring `PathMatcher`（路径分隔符为冒号`:`）以确定实例是否处理该消息。使用上述示例，“/ bus / refresh？ destination = customers： **”

将针对“客户”服务的所有实例，而不管配置文件和端口设置为 `ApplicationContext`ID。

应用程序上下文 ID 必须是唯一的

总线尝试从原始 `ApplicationEvent` 一次消除处理事件两次，一次从队列中消除。为此，它会检查发送应用程序上下文 id，以重新显示当前的应用程序上下文 ID。如果服务的多个实例具有相同的应用程序上下文 id，则不会处理事件。在本地机器上运行，每个服务将在不同的端口上，这将是应用程序上下文 ID 的一部分。Cloud Foundry 提供了区分的索引。要确保应用程序上下文 ID 是唯一的，请将 `spring.application.index` 设置为服务的每个实例唯一的值。例如，在 lattice 中，在 application.properties 中设置 `spring.application.index=${INSTANCE_INDEX}`（如果使用 configserver，请设置 bootstrap.properties）。

自定义 Message Broker

Spring Cloud Bus 使用 Spring Cloud Stream 广播消息，以便获取消息流，只需要在类路径中包含您选择的 binder 实现。有 AMQP（RabbitMQ）和 Kafka（`spring-cloud-starter-bus-[amqp,kafka]`）的公共汽车专用起动方便。一般来说，Spring Cloud Stream 依赖于用于配置中间件的 Spring Boot 自动配置约定，因此例如 AMQP 代理地址可以使用 `spring.rabbitmq.*`配置属性

更改。Spring Cloud Bus 在 `spring.cloud.bus.*`中具有少量本地配置属性（例如 `spring.cloud.bus.destination` 是使用外部中间件的主题的名称）。通常，默认值就足够了。

要更多地了解如何自定义消息代理设置，请参阅 Spring Cloud Stream 文档。

跟踪 Bus Events

可以通过设置 `spring.cloud.bus.trace.enabled=true` 来跟踪总线事件（`RemoteApplicationEvent` 的子类）。如果这样做，那么 Spring Boot `TraceRepository`（如果存在）将显示每个发送的事件和来自每个服务实例的所有 ack。示例（来自`/trace` 端点）：

```
{
  "timestamp": "2015-11-26T10:24:44.411+0000",
  "info": {
    "signal": "spring.cloud.bus.ack",
    "type": "RefreshRemoteApplicationEvent",
    "id": "c4d374b7-58ea-4928-a312-31984def293b",
    "origin": "stores:8081",
    "destination": "*:**"
  }
  },
  {
  "timestamp": "2015-11-26T10:24:41.864+0000",
  "info": {
    "signal": "spring.cloud.bus.sent",
    "type": "RefreshRemoteApplicationEvent",
    "id": "c4d374b7-58ea-4928-a312-31984def293b",
    "origin": "customers:9000",
    "destination": "*:**"
  }
  },
  {
```

```
  "timestamp": "2015-11-26T10:24:41.862+0000",
  "info": {
    "signal": "spring.cloud.bus.ack",
    "type": "RefreshRemoteApplicationEvent",
    "id": "c4d374b7-58ea-4928-a312-31984def293b",
    "origin": "customers:9000",
    "destination": "*:**"
  }
}
```

该跟踪显示 `RefreshRemoteApplicationEvent` 从 `customers:9000` 发送到所有服务，并且已被 `customers:9000` 和 `stores:8081` 收到（acked）。

为了处理信号，您可以向您的应用添加 `AckRemoteApplicationEvent` 和 `SentApplicationEvent` 类型的`@EventListener`（并启用跟踪）。或者您可以利用 `TraceRepository` 并从中挖掘数据。

注意	任何总线应用程序都可以跟踪 ack，但有时在一个可以对数据进行更复杂查询的中央 将其转发到专门的跟踪服务。

广播自己的 Events

总线可以携带任何类型为 `RemoteApplicationEvent` 的事件，但默认传输是 JSON，并且解串器需要知道哪些类型将提前使用。要注册一个新类型，它需要在 `org.springframework.cloud.bus.event` 的子包中。

要自定义事件名称，您可以在自定义类上使用`@JsonTypeName`，或者依赖默认策略来使用类的简单名称。请注意，生产者和消费者都需要访问类定义。

在自定义包中注册事件

如果您不能或不想为自定义事件使用 `org.springframework.cloud.bus.event` 的子包，则必须使用 `@RemoteApplicationEventScan` 指定要扫描类型为 `RemoteApplicationEvent` 的事件的包。使用 `@RemoteApplicationEventScan` 指定的软件包包括子包。

例如，如果您有一个名为 `FooEvent` 的自定义事件：

```
package com.acme;

public class FooEvent extends RemoteApplicationEvent {
    ...
}
```

您可以通过以下方式与解串器注册此事件：

```
package com.acme;

@Configuration
@RemoteApplicationEventScan
public class BusConfiguration {
    ...
}
```

没有指定一个值，使用`@RemoteApplicationEventScan` 的类的包将被注册。

在这个例子中，`com.acme` 将使用 `BusConfiguration` 的包进行注册。

您还可以使用`@RemoteApplicationEventScan`上的`value`，`basePackages`或`basePackageClasses`属性明确指定要扫描的软件包。例如：

```
package com.acme;

@Configuration
//@RemoteApplicationEventScan({"com.acme", "foo.bar"})
//@RemoteApplicationEventScan(basePackages = {"com.acme",
"foo.bar", "fizz.buzz"})
@RemoteApplicationEventScan(basePackageClasses =
BusConfiguration.class)
public class BusConfiguration {
    ...
}
```

以上`@RemoteApplicationEventScan`的所有示例都是等效的，因为`com.acme`程序包将通过在`@RemoteApplicationEventScan`上明确指定程序包来注册。请注意，您可以指定要扫描的多个基本软件包。

Spring Cloud Sleuth

Adrian Cole，Spencer Gibb，Marcin Grzejszczak，Dave Syer

Dalston.RELEASE

Spring Cloud Sleuth 为 Spring Cloud 实现分布式跟踪解决方案。

术语

Spring Cloud Sleuth 借用了 Dapper 的术语。

Span：工作的基本单位 例如，发送 RPC 是一个新的跨度，以及向 RPC 发送响应。Span 由跨度的唯一 64 位 ID 标识，跨度是其中一部分的跟踪的另一个 64 位 ID。跨度还具有其他数据，例如描述，时间戳记事件，键值注释（标签），导致它们的跨度的 ID 以及进程 ID（通常是 IP 地址）。

跨距开始和停止，他们跟踪他们的时间信息。创建跨度后，必须在将来的某个时刻停止。

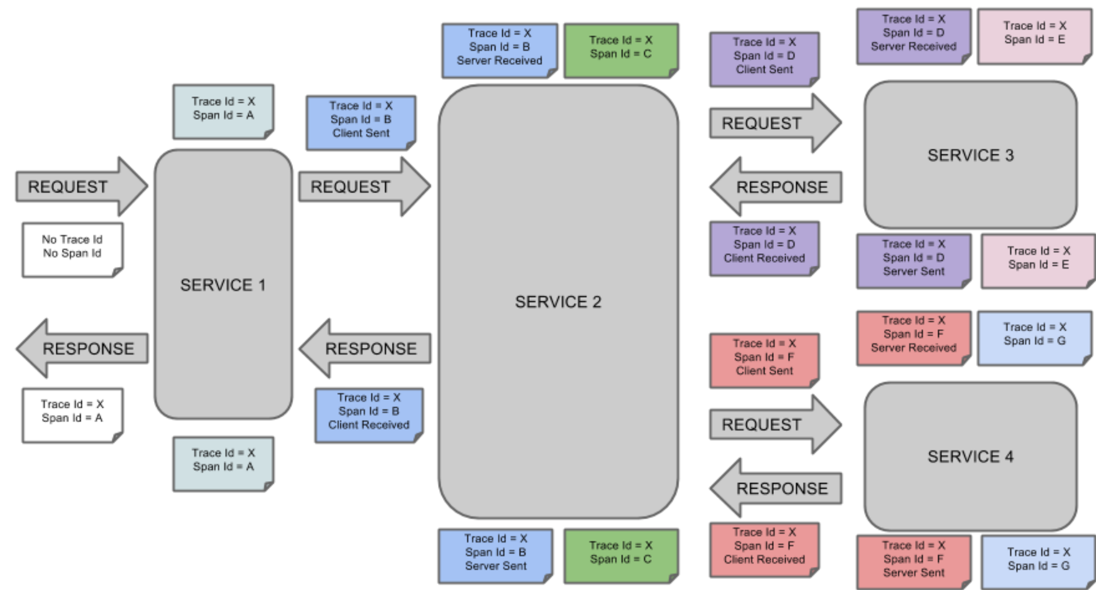
提示	启动跟踪的初始范围称为`root span`。该跨度的跨度 id 的值等于跟踪 ID。

跟踪：一组 spans 形成树状结构。例如，如果您正在运行分布式大数据存储，则可能会由 put 请求形成跟踪。

注释： 用于及时记录事件的存在。用于定义请求的开始和停止的一些核心注释是:

- **cs** - 客户端发送 - 客户端已经发出请求。此注释描绘了跨度的开始。
- **sr** - 服务器接收 - 服务器端得到请求，并将开始处理它。如果从此时间戳中减去 cs 时间戳，则会收到网络延迟。
- **ss** - 服务器发送 - 在完成请求处理后（响应发送回客户端时）注释。如果从此时间戳中减去 sr 时间戳，则会收到服务器端处理请求所需的时间。
- **cr** - 客户端接收 - 表示跨度的结束。客户端已成功接收到服务器端的响应。如果从此时间戳中减去 cs 时间戳，则会收到客户端从服务器接收响应所需的整个时间。

可视化 **Span** 和 **Trace** 将与 Zipkin 注释一起查看系统：

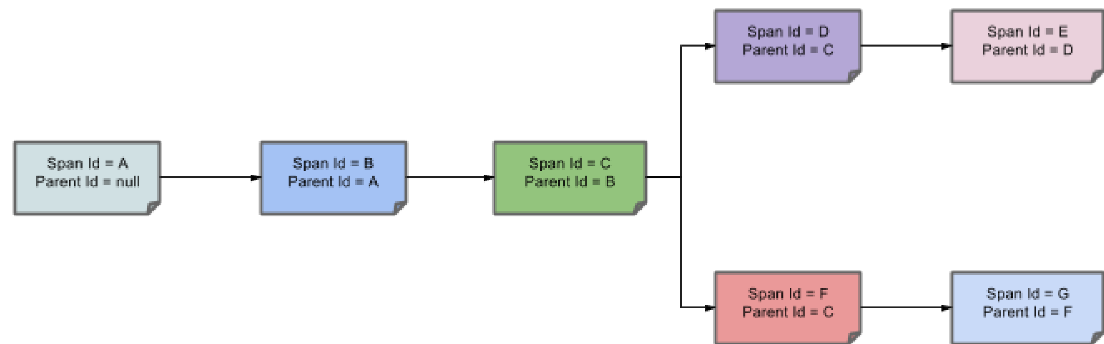


一个音符的每个颜色表示跨度（7 spans - 从 **A** 到 **G**）。如果您在笔记中有这样的信息：

```
Trace Id = X
Span Id = D
Client Sent
```

这意味着，当前的跨度**痕量-ID** 设置为 **X**，**Span -编号**设置为 **ð**。它也发出了 **客户端发送的**事件。

这样，spans 的父/子关系的可视化将如下所示：

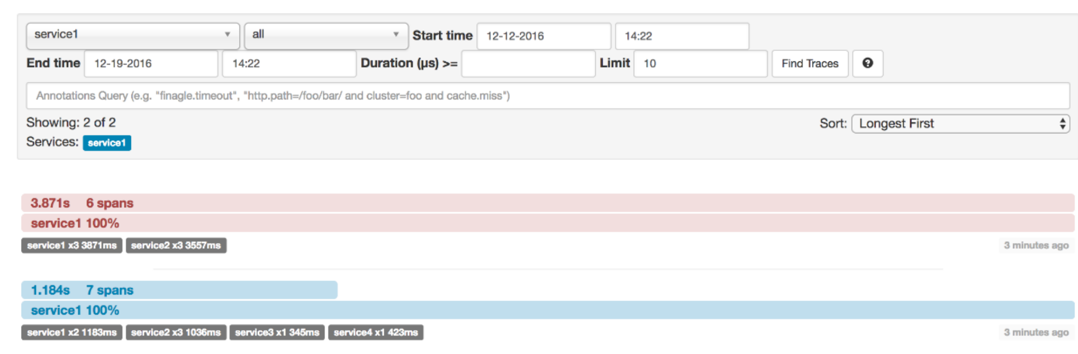


目的

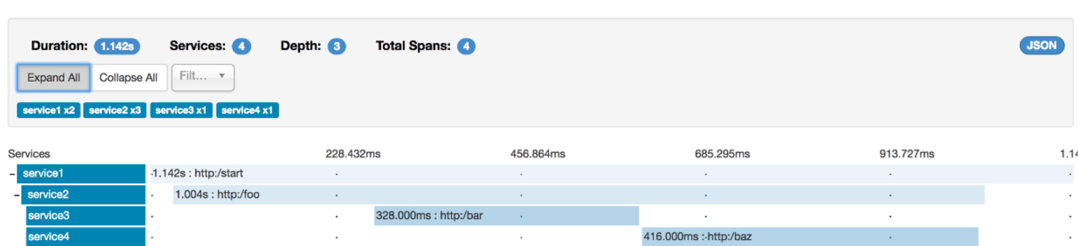
在以下部分中，将考虑上述图像中的示例。

分布式跟踪与 Zipkin

共有 **7 个 spans**。如果您在 Zipkin 中查看痕迹，您将在第二个曲目中看到这个数字：



但是，如果您选择特定的跟踪，那么您将看到 **4 spans**：



注意	当选择特定的跟踪时，您将看到合并的 spans。这意味着如果发送到服务器接收和服释的 Zipkin 有 2 个 spans，那么它们将被显示为一个跨度。

为什么在这种情况下，7 和 4 spans 之间有区别？

- 2 spans 来自 `http:/start` 范围。它具有服务器接收（SR）和服务器发送（SS）注释。
- 2 spans 来自 `service1` 到 `service2` 到 `http:/foo` 端点的 RPC 呼叫。它在 `service1` 方面具有客户端发送（CS）和客户端接收（CR）注释。它还在 `service2` 方面具有服务器接收（SR）和服务器发送（SS）注释。在物理上有 2 个 spans，但它们形成与 RPC 调用相关的 1 个逻辑跨度。
- 2 spans 来自 `service2` 到 `service3` 到 `http:/bar` 端点的 RPC 呼叫。它在 `service2` 方面具有客户端发送（CS）和客户接收（CR）注释。它还具有 `service3` 端的服务器接收（SR）和服务器发送（SS）注释。在物理上有 2 个 spans，但它们形成与 RPC 调用相关的 1 个逻辑跨度。
- 2 spans 来自 `service2` 到 `service4` 到 `http:/baz` 端点的 RPC 呼叫。它在 `service2` 方面具有客户端发送（CS）和客户接收（CR）注释。它还在 `service4` 侧具有服务器接收（SR）和服务器发送（SS）注释。在物理上有 2 个 spans，但它们形成与 RPC 调用相关的 1 个逻辑跨度。

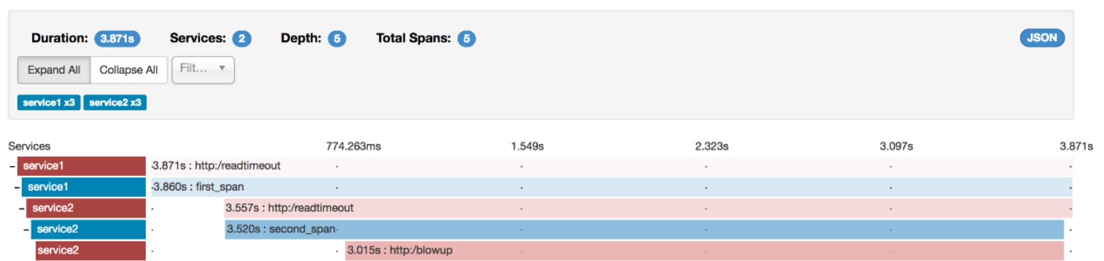
因此，如果我们计算 spans， `http:/start` 中有 **1 个**来自 `service1` 的呼叫 `service2`，**2** （`service2`）呼叫 `service3` 和 **2** （`service2`） `service4`。共 **7 个** spans。

逻辑上，我们看到 **Total Spans 的信息：4**，因为我们有 **1 个**跨度与传入请求相关的 `service1` 和 **3** spans 与 RPC 调用相关。

可视化错误

Zipkin 允许您可视化跟踪中的错误。当异常被抛出并且没有被捕获时，我们在 Zipkin 可以正确着色的跨度上设置适当的标签。您可以在痕迹列表中看到一条是红色的痕迹。这是因为抛出了一个异常。

如果您点击该轨迹，您将看到类似的图片



然后，如果您点击其中一个 spans，您将看到以下内容

service2.http:/readtimeout: 3.557s

AKA: service1,service2

Date Time	Relative Time	Annotation	Address
19/12/2016, 14:19:23	307.000ms	Client Send	127.0.0.1:8081 (service1)
19/12/2016, 14:19:23	310.000ms	Server Receive	127.0.0.1:8082 (service2)
19/12/2016, 14:19:26	3.836s	Server Send	127.0.0.1:8082 (service2)
19/12/2016, 14:19:27	3.864s	Client Receive	127.0.0.1:8081 (service1)

Key	Value
error	Request processing failed; nested exception is org.springframework.web.client.ResourceAccessException: I/O error on GET request for "http://localhost:8082/blowup": Read timed out; nested exception is java.net.SocketTimeoutException: Read timed out
http.host	localhost
http.method	GET
http.path	/readtimeout
http.status_code	500
http.url	http://localhost:8082/readtimeout
mvc.controller.class	BasicErrorController
mvc.controller.method	error

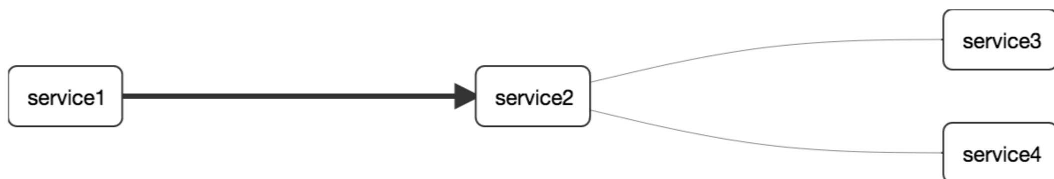
你可以看到，你可以很容易的看到错误的原因和整个 stacktrace 相关的。

实例



点击 Pivotal Web Services 图标即可实时查看！点击 Pivotal Web Services 图标直播！

Zipkin 中的依赖图将如下所示：





点击 Pivotal Web Services 图标即可实时查看！点击 Pivotal Web Services 图标直播！

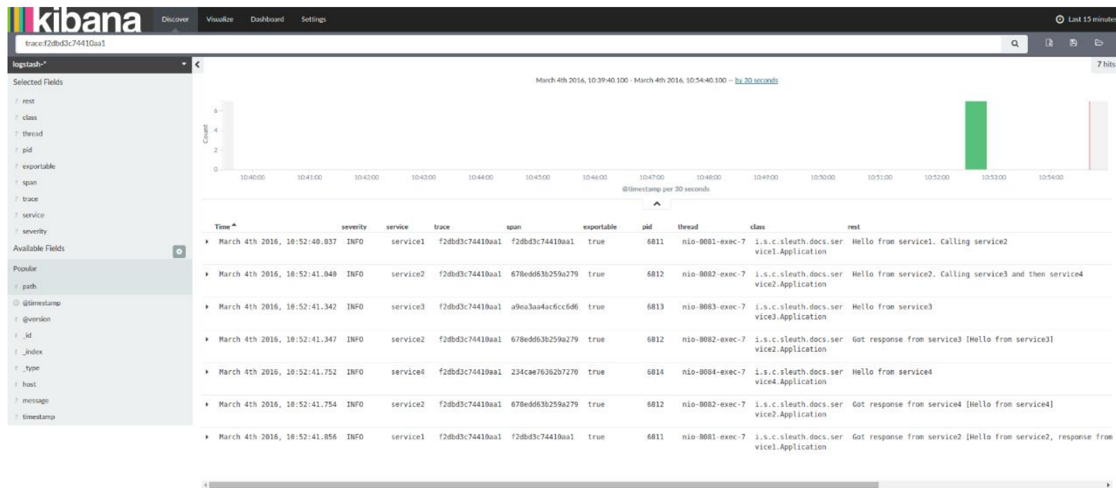
对数相关

当通过跟踪 id 等于例如 `2485ec27856c56f4` 来对这四个应用程序的日志进行灰名单时，将会得到以下内容：

```
service1.log:2016-02-26 11:15:47.561  INFO
[service1,2485ec27856c56f4,2485ec27856c56f4,true] 68058 -
-- [nio-8081-exec-1]
i.s.c.sleuth.docs.service1.Application   : Hello from
service1. Calling service2
service2.log:2016-02-26 11:15:47.710  INFO
[service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 -
-- [nio-8082-exec-1]
```

```
i.s.c.sleuth.docs.service2.Application   : Hello from
service2. Calling service3 and then service4
service3.log:2016-02-26 11:15:47.895  INFO
[service3,2485ec27856c56f4,1210be13194bfe5,true] 68060 ---
- [nio-8083-exec-1]
i.s.c.sleuth.docs.service3.Application   : Hello from
service3
service2.log:2016-02-26 11:15:47.924  INFO
[service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 -
-- [nio-8082-exec-1]
i.s.c.sleuth.docs.service2.Application   : Got response
from service3 [Hello from service3]
service4.log:2016-02-26 11:15:48.134  INFO
[service4,2485ec27856c56f4,1b1845262ffba49d,true] 68061 -
-- [nio-8084-exec-1]
i.s.c.sleuth.docs.service4.Application   : Hello from
service4
service2.log:2016-02-26 11:15:48.156  INFO
[service2,2485ec27856c56f4,9aa10ee6fbde75fa,true] 68059 -
-- [nio-8082-exec-1]
i.s.c.sleuth.docs.service2.Application   : Got response
from service4 [Hello from service4]
service1.log:2016-02-26 11:15:48.182  INFO
[service1,2485ec27856c56f4,2485ec27856c56f4,true] 68058 -
-- [nio-8081-exec-1]
i.s.c.sleuth.docs.service1.Application   : Got response
from service2 [Hello from service2, response from
service3 [Hello from service3] and from service4 [Hello
from service4]]
```

如果你使用像一个日志聚合工具 Kibana， Splunk 的等您可以订购所发生的事件。基巴纳的例子如下所示:



如果你想使用 Logstash，这里是 Logstash 的 Grok 模式：

```
filter {
      # pattern matching logback pattern
      grok {
            match => { "message" =>
"%{TIMESTAMP_ISO8601:timestamp}\s+%{LOGLEVEL:severity}\s+\[%{DATA:service},%{DATA:trace},%{DATA:span},%{DATA:exportable}\]\s+%{DATA:pid}\s+---\s+\[%{DATA:thread}\]\s+%{DATA:class}\s+:\s+%{GREEDYDATA:rest}" }
      }
}
```

注意 如果您想将 Grok 与 Cloud Foundry 的日志一起使用，则必须使用此模式：

```
filter {
      # pattern matching logback pattern
      grok {
            match => { "message" =>
"(?m)OUT\s+%{TIMESTAMP_ISO8601:timestamp}\s+%{LOGLEVEL:severity}\s+\[%{DATA:service},%{DATA:trace},%{DATA:span},%{DATA:exportable}\]\s+%{DATA:pid}\s+---\s+\[%{DATA:thread}\]\s+%{DATA:class}\s+:\s+%{GREEDYDATA:rest}" }
      }
}
```

使用 Logstash 进行 JSON 回溯

通常，您不希望将日志存储在文本文件中，而不是将 Logstash 可以立即选择的 JSON 文件中存储。为此，您必须执行以下操作（为了可读性，我们将依赖关系传递给 `groupId:artifactId:version` 符号。

依赖关系设置

- 确保 Logback 位于类路径（`ch.qos.logback:logback-core`）
- 添加 Logstash Logback 编码 - 版本 `4.6` 的示例：`net.logstash.logback:logstash-logback-encoder:4.6`

回读设置

您可以在下面找到一个 Logback 配置（名为 logback-spring.xml）的示例：

- 将来自应用程序的信息以 JSON 格式记录到 `build/${spring.application.name}.json` 文件
- 已经评论了两个额外的追加者 - 控制台和标准日志文件
- 具有与上一节所述相同的记录模式

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
	<include
resource="org/springframework/boot/logging/logback/defaul
ts.xml"/>

	<springProperty scope="context"
name="springAppName" source="spring.application.name"/>
	<!-- Example for logging into the build folder of
your project -->
	<property name="LOG_FILE" value="${BUILD_FOLDER:-
build}/${springAppName}"/>
```

```
	<!-- You can override this to have a custom pattern
-->
	<property name="CONSOLE_LOG_PATTERN"
	          value="%clr(%d{yyyy-MM-dd
HH:mm:ss.SSS}){faint} %clr(${LOG_LEVEL_PATTERN:-%5p}) %cl
r(${PID:- }){magenta} %clr(--
-){faint} %clr([%15.15t]){faint} %clr(%-
40.40logger{39}){cyan} %clr(:){faint} %m%n${LOG_EXCEPTION
_CONVERSION_WORD:-%wEx}"/>

	<!-- Appender to log to console -->
	<appender name="console"
class="ch.qos.logback.core.ConsoleAppender">
		<filter
class="ch.qos.logback.classic.filter.ThresholdFilter">
			<!-- Minimum logging level to be
presented in the console logs-->
			<level>DEBUG</level>
		</filter>
		<encoder>

	<pattern>${CONSOLE_LOG_PATTERN}</pattern>
			<charset>utf8</charset>
		</encoder>
	</appender>

	<!-- Appender to log to file -->
	<appender name="flatfile"
class="ch.qos.logback.core.rolling.RollingFileAppender">
		<file>${LOG_FILE}</file>
		<rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy
">
			<fileNamePattern>${LOG_FILE}.%d{yyyy-
MM-dd}.gz</fileNamePattern>
			<maxHistory>7</maxHistory>
		</rollingPolicy>
		<encoder>

	<pattern>${CONSOLE_LOG_PATTERN}</pattern>
			<charset>utf8</charset>
		</encoder>
	</appender>
```

```
      <!-- Appender to log to file in a JSON format -->
      <appender name="logstash"
class="ch.qos.logback.core.rolling.RollingFileAppender">
            <file>${LOG_FILE}.json</file>
            <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy
">

      <fileNamePattern>${LOG_FILE}.json.%d{yyyy-MM-
dd}.gz</fileNamePattern>
                  <maxHistory>7</maxHistory>
            </rollingPolicy>
            <encoder
class="net.logstash.logback.encoder.LoggingEventComposite
JsonEncoder">
                  <providers>
                        <timestamp>
                              <timeZone>UTC</timeZone>
                        </timestamp>
                        <pattern>
                              <pattern>
                                    {
                                    "severity":
"%level",
                                    "service":
"${springAppName:-}",
                                    "trace": "%X{X-B3-
TraceId:-}",
                                    "span": "%X{X-B3-
SpanId:-}",
                                    "parent": "%X{X-B3-
ParentSpanId:-}",
                                    "exportable":
"%X{X-Span-Export:-}",
                                    "pid": "${PID:-}",
                                    "thread":
"%thread",
                                    "class":
"%logger{40}",
                                    "rest": "%message"
                                    }
                              </pattern>
                        </pattern>
                  </providers>
```

```
                </encoder>
        </appender>

        <root level="INFO">
                <appender-ref ref="console"/>
                <!-- uncomment this to have also JSON logs -
->
                <!--<appender-ref ref="logstash"/>-->
                <!--<appender-ref ref="flatfile"/>-->
        </root>
</configuration>
```

注意	如果您使用自定义 `logback-spring.xml`，则必须通过 `bootstrap` `applicati` 递 `spring.application.name`。否则您的自定义 logback 文件将不会正确读取该

传播 Span 上下文

跨度上下文是必须传播到任何子进程跨越进程边界的状态。Span 背景的一部分是行李。跟踪和跨度 ID 是跨度上下文的必需部分。行李是可选的部分。

行李是一组密钥：存储在范围上下文中的值对。行李与痕迹一起旅行，并附在每一个跨度上。Spring Cloud 如果 HTTP 标头以 `baggage-` 为前缀，并且以 `baggage_` 开头的消息传递，Sleuth 将会明白标题是行李相关的。

重要	行李物品的数量或大小目前没有限制。但是，请记住，太多可能会降低系统吞吐量或超出了传输级消息或报头容量，可能会使应用程序崩溃。

在跨度上设置行李的示例：

```
Span initialSpan = this.tracer.createSpan("span");
initialSpan.setBaggageItem("foo", "bar");
```

行李与 Span 标签

行李随行旅行（即每个孩子跨度都包含其父母的行李）。Zipkin 不了解行李，甚至不会收到这些信息。

标签附加到特定的跨度 - 它们仅针对该特定跨度呈现。但是，您可以通过标签搜索查找跟踪，其中存在具有搜索标签值的跨度。

如果您希望能够根据行李查找跨度，则应在根跨度中添加相应的条目作为标签。

```
@Autowired Tracer tracer;

Span span = tracer.getCurrentSpan();
String baggageKey = "key";
String baggageValue = "foo";
span.setBaggageItem(baggageKey, baggageValue);
tracer.addTag(baggageKey, baggageValue);
```

添加到项目中

只有 Sleuth（对数相关）

如果您只想从 Spring Cloud Sleuth 中获利，而没有 Zipkin 集成，只需将 `spring-cloud-starter-sleuth` 模块添加到您的项目中即可。

Maven 的

```
<dependencyManagement> (1)
        <dependencies>
            <dependency>

<groupId>org.springframework.cloud</groupId>
              <artifactId>spring-cloud-
dependencies</artifactId>
              <version>Camden.RELEASE</version>
              <type>pom</type>
              <scope>import</scope>
```

```
            </dependency>
        </dependencies>
   </dependencyManagement>

   <dependency> (2)
       <groupId>org.springframework.cloud</groupId>
       <artifactId>spring-cloud-starter-
sleuth</artifactId>
   </dependency>
```

1. 为了不自己选择版本，如果您通过 Spring BOM 添加依赖关系管理，会更好

2. 将依赖关系添加到 `spring-cloud-starter-sleuth`

摇篮

```
dependencyManagement { (1)
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-
dependencies:Camden.RELEASE"
    }
}

dependencies { (2)
    compile "org.springframework.cloud:spring-cloud-
starter-sleuth"
}
```

1. 为了不自己选择版本，如果您通过 Spring BOM 添加依赖关系管理，会更好

2. 将依赖关系添加到 `spring-cloud-starter-sleuth`

通过 HTTP 访问 Zipkin

如果你想要 Sleuth 和 Zipkin 只需添加 `spring-cloud-starter-zipkin` 依赖关系。

Maven 的

```
<dependencyManagement> (1)
        <dependencies>
```

```
            <dependency>

<groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-
dependencies</artifactId>
                <version>Camden.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
   </dependencyManagement>

   <dependency> (2)
       <groupId>org.springframework.cloud</groupId>
       <artifactId>spring-cloud-starter-
zipkin</artifactId>
   </dependency>
```

1. 为了不自己选择版本，如果您通过 Spring BOM 添加依赖关系管理，会更好

2. 将依赖关系添加到 `spring-cloud-starter-zipkin`

摇篮

```
dependencyManagement { (1)
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-
dependencies:Camden.RELEASE"
    }
}

dependencies { (2)
    compile "org.springframework.cloud:spring-cloud-
starter-zipkin"
}
```

1. 为了不自己选择版本，如果您通过 Spring BOM 添加依赖关系管理，会更好

2. 将依赖关系添加到 `spring-cloud-starter-zipkin`

通过 Spring Cloud Stream 使用 Zipkin 的 Sleuth

如果你想要 Sleuth 和 Zipkin 只需添加 `spring-cloud-sleuth-stream` 依赖关系。

Maven 的

```
<dependencyManagement> (1)
        <dependencies>
            <dependency>

<groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-
dependencies</artifactId>
                <version>Camden.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
   </dependencyManagement>

   <dependency> (2)
       <groupId>org.springframework.cloud</groupId>
       <artifactId>spring-cloud-sleuth-stream</artifactId>
   </dependency>
   <dependency> (3)
       <groupId>org.springframework.cloud</groupId>
       <artifactId>spring-cloud-starter-
sleuth</artifactId>
   </dependency>
   <!-- EXAMPLE FOR RABBIT BINDING -->
   <dependency> (4)
       <groupId>org.springframework.cloud</groupId>
       <artifactId>spring-cloud-stream-binder-
rabbit</artifactId>
   </dependency>
```

1. 为了不自己选择版本，如果您通过 Spring BOM 添加依赖关系管理，会更好
2. 将依赖关系添加到 `spring-cloud-sleuth-stream`
3. 将依赖关系添加到 `spring-cloud-starter-sleuth` 中，这样就可以下载依赖关系

4. 添加一个粘合剂（例如 Rabbit binder）来告诉 Spring Cloud Stream 应该绑定什么

摇篮

```
dependencyManagement { (1)
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:Camden.RELEASE"
    }
}

dependencies {
    compile "org.springframework.cloud:spring-cloud-sleuth-stream" (2)
    compile "org.springframework.cloud:spring-cloud-starter-sleuth" (3)
    // Example for Rabbit binding
    compile "org.springframework.cloud:spring-cloud-stream-binder-rabbit" (4)
}
```

1. 为了不自己选择版本，如果您通过 Spring BOM 添加依赖关系管理，会更好
2. 将依赖关系添加到 `spring-cloud-sleuth-stream`
3. 将依赖关系添加到 `spring-cloud-starter-sleuth` 中，这样就可以下载所有依赖关系
4. 添加一个粘合剂（例如 Rabbit binder）来告诉 Spring Cloud Stream 应该绑定什么

Spring Cloud Sleuth Stream Zipkin 收藏家

如果要启动 Spring Cloud Sleuth Stream Zipkin 收藏夹，只需添加 `spring-cloud-sleuth-zipkin-stream` 依赖关系即可

Maven 的

```
<dependencyManagement> (1)
        <dependencies>
            <dependency>

<groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-
dependencies</artifactId>
                <version>Camden.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
   </dependencyManagement>

   <dependency> (2)
       <groupId>org.springframework.cloud</groupId>
       <artifactId>spring-cloud-sleuth-zipkin-
stream</artifactId>
   </dependency>
   <dependency> (3)
       <groupId>org.springframework.cloud</groupId>
       <artifactId>spring-cloud-starter-
sleuth</artifactId>
   </dependency>
   <!-- EXAMPLE FOR RABBIT BINDING -->
   <dependency> (4)
       <groupId>org.springframework.cloud</groupId>
       <artifactId>spring-cloud-stream-binder-
rabbit</artifactId>
   </dependency>
```

1. 为了不自己选择版本，如果您通过 Spring BOM 添加依赖关系管理，会更好
2. 将依赖关系添加到 `spring-cloud-sleuth-zipkin-stream`
3. 将依赖关系添加到 `spring-cloud-starter-sleuth` - 这样一来，所有的依赖依赖将被下载

4. 添加一个粘合剂（例如 Rabbit binder）来告诉 Spring Cloud Stream 应该绑定什么

摇篮

```
dependencyManagement { (1)
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:Camden.RELEASE"
    }
}

dependencies {
    compile "org.springframework.cloud:spring-cloud-sleuth-zipkin-stream" (2)
    compile "org.springframework.cloud:spring-cloud-starter-sleuth" (3)
    // Example for Rabbit binding
    compile "org.springframework.cloud:spring-cloud-stream-binder-rabbit" (4)
}
```

1. 为了不自己选择版本，如果您通过 Spring BOM 添加依赖关系管理，会更好
2. 将依赖关系添加到 `spring-cloud-sleuth-zipkin-stream`
3. 将依赖关系添加到 `spring-cloud-starter-sleuth` - 这样将依赖关系依赖下载
4. 添加一个粘合剂（例如 Rabbit binder）来告诉 Spring Cloud Stream 应该绑定什么

然后使用`@EnableZipkinStreamServer`注释注释你的主类：

```
package example;

import org.springframework.boot.SpringApplication;
```

```
import
org.springframework.boot.autoconfigure.SpringBootApplicat
ion;
import
org.springframework.cloud.sleuth.zipkin.stream.EnableZipk
inStreamServer;

@SpringBootApplication
@EnableZipkinStreamServer
public class ZipkinStreamServerApplication {

	public static void main(String[] args) throws
Exception {

	SpringApplication.run(ZipkinStreamServerApplication
.class, args);
	}

}
```

额外的资源

Marcin Grzejszczak 谈论 Spring Cloud Sleuth 和 Zipkin

点击此处查看视频

特征

- 将跟踪和跨度添加到 Slf4J MDC，以便您可以从日志聚合器中的给定跟踪或跨度中提取所有日志。示例日志：

```
2016-02-02 15:30:57.902  INFO
[bar,6bfd228dc00d216b,6bfd228dc00d216b,false] 23030 ---
[nio-8081-exec-3] ...
```

- ```
  2016-02-02 15:30:58.372 ERROR
  [bar,6bfd228dc00d216b,6bfd228dc00d216b,false] 23030 ---
  [nio-8081-exec-3] ...
  2016-02-02 15:31:01.936  INFO
  [bar,46ab0d418373cbc9,46ab0d418373cbc9,false] 23030 ---
  [nio-8081-exec-4] ...
  ```
  注意 MDC 中的`[appname,traceId,spanId,exportable]`条目：
  - **spanId** - 发生特定操作的 ID
  - **appname** - 记录跨度的应用程序的名称
  - **traceId** - 包含跨度的延迟图的 ID
  - **导出** -日志是否应该被导出到 Zipkin 与否。你什么时候希望跨度不能出口？在这种情况下，你想在 Span 中包装一些操作，并将它写入日志。
- 提供对共同分布式跟踪数据模型的抽象：trace，spans（形成 DAG），注释，键值注释。松散地基于 HTrace，但 Zipkin（Dapper）兼容。
- Sleuth 记录定时信息以辅助延迟分析。使用窃贼，您可以精确定位应用程序中的延迟原因。Sleuth 被写入不会记录太多，并且不会导致您的生产应用程序崩溃。
  - 传播有关您的呼叫图表带内的结构数据，其余的是带外。
  - 包括有意见的层次测试，如 HTTP
  - 包括采样策略来管理卷
  - 可以向 Zipkin 系统报告查询和可视化
- 仪器来自 Spring 应用程序（servlet 过滤器，异步终结点，休息模板，调度操作，消息通道，zuul 过滤器，假客户端）的通用入口和出口点。
  ```

- Sleuth 包括在 http 或消息传递边界上加入跟踪的默认逻辑。例如，http 传播通过 Zipkin 兼容的请求标头工作。该传播逻辑是通过 `SpanInjector` 和 `SpanExtractor` 实现来定义和定制的。
- Sleuth 可以在进程之间传播上下文（也称为行李）。这意味着如果您设置了 Span 行李元素，那么它将通过 HTTP 或消息传递到其他进程发送到下游。
- 提供创建/继续 spans 并通过注释添加标签和日志的方法。
- 提供接受/删除 spans 的简单指标。
- 如果 `spring-cloud-sleuth-zipkin`，则应用程序将生成并收集 Zipkin 兼容的跟踪。默认情况下，它通过 HTTP 将其发送到 localhost 上的 Zipkin 服务器（端口 9411）。使用 `spring.zipkin.baseUrl` 配置服务的位置。
- 如果 `spring-cloud-sleuth-stream`，则该应用将通过 [Spring Cloud Stream] 生成和收集跟踪。您的应用程序自动成为通过您的代理商发送的跟踪消息的生产者（例如 RabbitMQ，Apache Kafka，Redis））。

重要	如果使用 Zipkin 或 Stream，请使用 `spring.sleuth.sampler.percentage`（默 比。否则你可能认为 Sleuth 不工作，因为它省略了一些 spans。
注意	始终设置 SLF4J MDC，并且 Logback 用户将立即按照上述示例查看日志中的跟踪和跟 己的格式化程序以获得相同的结果。默认值为 `logging.pattern.level` 设置为`%` `[${spring.zipkin.service.name:${spring.application.name:-}},%X` `SpanId:-},%X{X-Span-Export:-}]`（这是回访用户的 Spring Boot 功能）。 自动应用此模式。

采样

在分布式跟踪中，数据量可能非常高，因此采样可能很重要（您通常不需要导出所有 spans 以获得正在发生的情况）。Spring Cloud Sleuth 具有 `Sampler` 策略，您可以实现该策略来控制采样算法。采样器不会停止生成跨度（相关）ids，但是它们确实阻止了附加和导出的标签和事件。默认情况下，您将获得一个策略，如果跨度已经处于活动状态，则会继续跟踪，但新策略始终被标记为不可导出。如果您的所有应用程序都使用此采样器运行，您将看到日志中的跟踪，但不会在任何远程存储中。对于测试，默认值通常是足够的，如果您仅使用日志（例如使用 ELK 聚合器），则可能是您需要的。如果要将 span 数据导出到 Zipkin 或 Spring Cloud Stream，则还有一个 `AlwaysSampler` 导出所有内容，并且 `PercentageBasedSampler` 对 spans 的固定分数进行采样。

注意	如果您使用 `spring-cloud-sleuth-zipkin` 或 `spring-cloud-sleuth-strea` `PercentageBasedSampler`。您可以使用 `spring.sleuth.sampler.percenta` 到 `1.0` 的双倍，所以不是百分比。为了向后兼容性原因，我们不会更改属性名称。

可以通过创建一个 bean 定义来安装采样器，例如：

```
@Bean
public Sampler defaultSampler() {
	return new AlwaysSampler();
}
```

提示	您可以将 HTTP 标头 `X-B3-Flags` 设置为 `1`，或者在进行消息传递时，您可以将 `sp` 样决定如何，当前跨度都将被强制输出。

仪表

Spring Cloud Sleuth 自动为您的所有 Spring 应用程序设备，因此您不必执行任何操作即可激活它。根据可用的堆栈，例如对于我们使用 `Filter` 的 servlet web 应用程序，以及 Spring Integration 我们使用 `ChannelInterceptors`），可以使用各种技术添加仪器。

您可以自定义 span 标签中使用的键。为了限制跨度数据量，默认情况下，HTTP 请求只会被标记为少量的元数据，如状态码，主机和 URL。您可以通过配置 `spring.sleuth.keys.http.headers`（头名称列表）来添加请求头。

注意	记住，如果有一个 `Sampler` 允许它（默认情况下没有，所以没有意外收集太多数据 集和导出标签。
注意	目前，Spring Cloud Sleuth 中的测试仪器是渴望的 - 这意味着我们正在积极地尝 没有将数据导出到跟踪系统的情况下，也会捕获定时事件。这种做法在将来可能会

Span 生命周期

您可以通过 org.springframework.cloud.sleuth.Tracer 接口在 Span 上**执行**以下操作：

- 开始 - 当您启动一个 span 时，它的名称被分配，并且记录开始时间戳。
- 关闭 - 跨度完成（记录跨度的结束时间），如果跨度可**导出，**则它将有资格收集到 Zipkin。该跨度也从当前线程中移除。
- 继续 - 将创建一个新的跨度实例，而它将是它继续的一个副本。
- 分离 - 跨度不会停止或关闭。它只从当前线程中删除。

- 使用显式父项创建 - 您可以创建一个新的跨度，并为其设置一个显式父级

提示	Spring 为您创建了一个 `Tracer` 的实例。为了使用它，你需要的只是自动连接它。

创建和关闭 spans

您可以使用 **Tracer** 界面手动创建 spans 。

```
// Start a span. If there was a span present in this
thread it will become
// the `newSpan`'s parent.
Span newSpan = this.tracer.createSpan("calculateTax");
try {
	// ...
	// You can tag a span
	this.tracer.addTag("taxValue", taxValue);
	// ...
	// You can log an event on a span
	newSpan.logEvent("taxCalculated");
} finally {
	// Once done remember to close the span. This will
allow collecting
	// the span to send it to Zipkin
	this.tracer.close(newSpan);
}
```

在这个例子中，我们可以看到如何创建一个新的跨度实例。假设这个线程中已经存在跨度，那么它将成为该跨度的父代。

重要	创建跨度后始终清洁！如果要将其发送到 Zipkin，请不要忘记关闭跨度。
重要	如果您的 span 包含的名称大于 50 个字符，则该名称将被截断为 50 个字符。你的名 迟问题，有时甚至引发异常。

继续 spans

有时你不想创建一个新的跨度，但你想继续。这种情况的例子可能是（当然这取决于用例）：

- **AOP** - 如果在达到方面之前已经创建了一个跨度，则可能不想创建一个新的跨度。
- **Hystrix** - 执行 Hystrix 命令很可能是当前处理的逻辑部分。实际上，它只是一个技术实现细节，你不一定要反映在跟踪中作为一个单独的存在。

持续的跨度实例等于它继续的范围：

```
Span continuedSpan =
this.tracer.continueSpan(spanToContinue);
assertThat(continuedSpan).isEqualTo(spanToContinue);
```

要继续跨度，您可以使用 **Tracer** 界面。

```
// let's assume that we're in a thread Y and we've
received
// the `initialSpan` from thread X
Span continuedSpan =
this.tracer.continueSpan(initialSpan);
try {
	// ...
	// You can tag a span
	this.tracer.addTag("taxValue", taxValue);
	// ...
	// You can log an event on a span
	continuedSpan.logEvent("taxCalculated");
} finally {
	// Once done remember to detach the span. That way
you'll
	// safely remove it from the current thread without
closing it
	this.tracer.detach(continuedSpan);
}
```

重要 | 创建跨度后始终清洁！如果在一个线程（例如线程 X）中开始了某些工作，并且正在
不要忘记分离跨距。那么线程 Y，Z 中的 spans 在工作结束时应该被分离。当收集结

用明确的父代创建 spans

您可能想要开始一个新的跨度，并提供该跨度的显式父级。假设跨度的父项在一个线程中，并且要在另一个线程中启动一个新的跨度。`Tracer` 接口的 `startSpan` 方法是您要查找的方法。

```
// let's assume that we're in a thread Y and we've
received
// the `initialSpan` from thread X. `initialSpan` will be
the parent
// of the `newSpan`
Span newSpan =
this.tracer.createSpan("calculateCommission",
initialSpan);
try {
	// ...
	// You can tag a span
	this.tracer.addTag("commissionValue",
commissionValue);
	// ...
	// You can log an event on a span
	newSpan.logEvent("commissionCalculated");
} finally {
	// Once done remember to close the span. This will
allow collecting
	// the span to send it to Zipkin. The tags and
events set on the
	// newSpan will not be present on the parent
	this.tracer.close(newSpan);
}
```

重要 | 创建这样一个跨度后，记得关闭它。否则，您将在您的日志中看到很多警告，其中
个跨度，而不是您要关闭的线程。更糟糕的是，您的 spans 不会正确关闭，因此不

命名 spans

选择一个跨度名称不是一件小事。Span 名称应该描述一个操作名称。名称应该是低基数（例如不包括标识符）。

由于有很多仪器仪表在一些跨度名称将是人为的：

- `controller-method-name` 当控制器以方法名 `conrollerMethodName` 接收时
- `async` 通过包装 `Callable` 和 `Runnable` 完成异步操作。
- `@Scheduled` 注释方法将返回类的简单名称。

幸运的是，对于异步处理，您可以提供明确的命名。

@SpanName 注释

您可以通过`@SpanName` 注释显式指定该跨度。

```
@SpanName("calculateTax")
class TaxCountingRunnable implements Runnable {

	@Override public void run() {
		// perform logic
	}
}
```

在这种情况下，以下列方式处理时：

```
Runnable runnable = new TraceRunnable(tracer, spanNamer,
new TaxCountingRunnable());
Future<?> future = executorService.submit(runnable);
// ... some additional logic ...
future.get();
```

该范围将被命名为`calculateTax`。

toString（）方法

为`Runnable`或`Callable`创建单独的课程很少见。通常，创建这些类的匿名实例。如果没有`@SpanName`注释，我们将检查该类是否具有`toString()`方法的自定义实现。

所以执行这样的代码：

```
Runnable runnable = new TraceRunnable(tracer, spanNamer,
new Runnable() {
	@Override public void run() {
		// perform logic
	}

	@Override public String toString() {
		return "calculateTax";
	}
});
Future<?> future = executorService.submit(runnable);
// ... some additional logic ...
future.get();
```

将导致创建一个名为`calculateTax`的跨度。

管理 spans 注释

合理

这个功能的主要论据是

- api-agnostic 意味着与跨度进行合作
  - 使用注释允许用户添加到跨度 api 没有库依赖的跨度。这允许 Sleuth 将其核心 api 的影响改变为对用户代码的影响较小。
- 减少基础跨度作业的表面积。
  - 没有这个功能，必须使用 span api，它具有不正确使用的生命周期命令。通过仅显示范围，标签和日志功能，用户可以协作，而不会意外中断跨度生命周期。
- 与运行时生成的代码协作
  - 使用诸如 Spring Data / Feign 的库，在运行时生成接口的实现，从而跨越对象的包装是乏味的。现在，您可以通过这些接口的接口和参数提供注释

创建新的 spans

如果您真的不想手动创建本地 spans，您可以从`@NewSpan`注释中获利。此外，我们还提供`@SpanTag`注释，以自动方式添加标签。

我们来看一些使用的例子。

```
@NewSpan
void testMethod();
```

注释没有任何参数的方法将导致创建名称将等于注释方法名称的新跨度。

```
@NewSpan("customNameOnTestMethod4")
void testMethod4();
```

如果您在注释中提供值（直接或通过 `name` 参数），则创建的范围将具有提供的值的名称。

```
// method declaration
@NewSpan(name = "customNameOnTestMethod5")
void testMethod5(@SpanTag("testTag") String param);

// and method execution
this.testBean.testMethod5("test");
```

您可以组合名称和标签。我们来关注后者。在这种情况下，无论注释方法的参数运行时值的值如何 - 这将是标记的值。在我们的示例中，标签密钥将为 `testTag`，标签值为 `test`。

```
@NewSpan(name = "customNameOnTestMethod3")
@Override
public void testMethod3() {
}
```

您可以将`@NewSpan`注释放在类和接口上。如果覆盖接口的方法并提供不同的`@NewSpan`注释值，则最具体的一个获胜（在这种情况下 `customNameOnTestMethod3` 将被设置）。

继续 spans

如果您只想添加标签和注释到现有的跨度，就可以使用如下所示的 `@ContinueSpan` 注释。请注意，与`@NewSpan` 注释相反，您还可以通过 `log` 参数添加日志：

```
// method declaration
@ContinueSpan(log = "testMethod11")
void testMethod11(@SpanTag("testTag11") String param);

// method execution
this.testBean.testMethod11("test");
```

这样，跨越将继续下去：

- 将创建名称为 `testMethod11.before` 和 `testMethod11.after` 的日志
- 如果抛出异常，也将创建一个日志 `testMethod11.afterFailure`
- 将创建密钥 `testTag11` 和值 `test` 的标签

更高级的标签设置

有三种不同的方法可以将标签添加到跨度。所有这些都由 `SpanTag` 注释控制。优先级是：

- 尝试使用 `TagValueResolver` 类型的 bean，并提供名称
- 如果没有提供 bean 名称，请尝试评估一个表达式。我们正在搜索一个 `TagValueExpressionResolver` bean。默认实现使用 SPEL 表达式解析。
- 如果没有提供任何表达式来评估只返回参数的 `toString()` 值

自定义提取器

以下方法的标签值将由 `TagValueResolver` 接口的实现来计算。其类名必须作为 `resolver` 属性的值传递。

有这样一个注释的方法：

```
@NewSpan
public void getAnnotationForTagValueResolver(@SpanTag(key
= "test", resolver = TagValueResolver.class) String test)
{
}
```

和这样一个 `TagValueResolver` bean 实现

```
@Bean(name = "myCustomTagValueResolver")
public TagValueResolver tagValueResolver() {
      return parameter -> "Value from
myCustomTagValueResolver";
}
```

将导致标签值的设置等于 `Value from myCustomTagValueResolver`。

解决表达式的价值

有这样一个注释的方法：

```
@NewSpan
public void
getAnnotationForTagValueExpression(@SpanTag(key = "test",
expression = "length() + ' characters'") String test) {
}
```

并且没有自定义的 `TagValueExpressionResolver` 实现将导致对 SPEL 表达式的评估，并且将在 span 上设置值为 `4 characters` 的标签。如果要使用其他表达式解析机制，您可以创建自己的 bean 实现。

使用 toString 方法

有这样一个注释的方法：

```
@NewSpan
public void
getAnnotationForArgumentToString(@SpanTag("test") Long
param) {
}
```

如果使用值为 `15` 执行，则将导致设置 String 值为`"15"`的标记。

自定义

感谢 `SpanInjector` 和 `SpanExtractor`，您可以自定义 spans 的创建和传播方式。

目前有两种在进程之间传递跟踪信息的内置方式：

- 通过 Spring Integration
- 通过 HTTP

Span ids 从 Zipkin 兼容（B3）头（`Message` 或 HTTP 头）中提取，以启动或加入现有跟踪。跟踪信息被注入到任何出站请求中，所以下一跳可以提取它们。

与以前版本的 Sleuth 相比，重要的变化是 Sleuth 正在实施 Open Tracing 的 `TextMap` 概念。在 Sleuth，它被称为 `SpanTextMap`。基本上这个想法是通过 `SpanTextMap` 可以抽象出任何通信手段（例如消息，http 请求等）。这个抽象定义了如何将数据插入到载体中以及如何从那里检索数据。感谢这样，如果您想要使用一个使用 `FooRequest` 作为发送 HTTP 请求的平均值的新 HTTP 库，那么您必须创建一个 `SpanTextMap` 的实现，它将调用委托给 `FooRequest` 检索和插入 HTTP 标头。

Spring Integration

对于 Spring Integration，有 2 个接口负责从 `Message` 创建 Span。这些是：

- `MessagingSpanTextMapExtractor`
- `MessagingSpanTextMapInjector`

您可以通过提供自己的实现来覆盖它们。

HTTP

对于 HTTP，有 2 个接口负责从 `Message` 创建 Span。这些是：

- `HttpSpanExtractor`
- `HttpSpanInjector`

您可以通过提供自己的实现来覆盖它们。

例

我们假设，而不是标准的 Zipkin 兼容的跟踪 HTTP 头名称

- for trace id - `correlationId`
- for span id - `mySpanId`

这是`SpanExtractor`的一个例子

```
static class CustomHttpSpanExtractor implements
HttpSpanExtractor {

	@Override public Span joinTrace(SpanTextMap
carrier) {
		Map<String, String> map =
TextMapUtil.asMap(carrier);
		long traceId =
Span.hexToId(map.get("correlationid"));
		long spanId =
Span.hexToId(map.get("myspanid"));
		// extract all necessary headers
		Span.SpanBuilder builder =
Span.builder().traceId(traceId).spanId(spanId);
		// build rest of the Span
		return builder.build();
	}
}

static class CustomHttpSpanInjector implements
HttpSpanInjector {

	@Override
	public void inject(Span span, SpanTextMap carrier)
{
		carrier.put("correlationId",
span.traceIdString());
		carrier.put("mySpanId",
Span.idToHex(span.getSpanId()));
	}
}
```

你可以这样注册:

```
@Bean
HttpSpanInjector customHttpSpanInjector() {
	return new CustomHttpSpanInjector();
```

```
}

@Bean
HttpSpanExtractor customHttpSpanExtractor() {
	return new CustomHttpSpanExtractor();
}
```

Spring Cloud 为了安全起见，Sleuth 不会将跟踪/跨度相关的标头添加到 Http 响应。如果您需要标题，那么将标题注入 Http 响应的自定义 `SpanInjector`，并且可以使用以下方式添加一个使用此标签的 Servlet 过滤器：

```
static class CustomHttpServletResponseSpanInjector
extends ZipkinHttpSpanInjector {

	@Override
	public void inject(Span span, SpanTextMap carrier)
{
		super.inject(span, carrier);
		carrier.put(Span.TRACE_ID_NAME,
span.traceIdString());
		carrier.put(Span.SPAN_ID_NAME,
Span.idToHex(span.getSpanId()));
	}
}

static class HttpResponseInjectingTraceFilter extends
GenericFilterBean {

	private final Tracer tracer;
	private final HttpSpanInjector spanInjector;

	public HttpResponseInjectingTraceFilter(Tracer
tracer, HttpSpanInjector spanInjector) {
		this.tracer = tracer;
		this.spanInjector = spanInjector;
	}

	@Override
	public void doFilter(ServletRequest request,
ServletResponse servletResponse, FilterChain filterChain)
throws IOException, ServletException {
```

你可以这样注册:

```
            HttpServletResponse response =
(HttpServletResponse) servletResponse;
            Span currentSpan =
this.tracer.getCurrentSpan();
            this.spanInjector.inject(currentSpan, new
HttpServletResponseTextMap(response));
            filterChain.doFilter(request, response);
      }

       class HttpServletResponseTextMap implements
SpanTextMap {

             private final HttpServletResponse delegate;


HttpServletResponseTextMap(HttpServletResponse delegate)
{
                   this.delegate = delegate;
             }

             @Override
             public Iterator<Map.Entry<String, String>>
iterator() {
                   Map<String, String> map = new
HashMap<>();
                   for (String header :
this.delegate.getHeaderNames()) {
                         map.put(header,
this.delegate.getHeader(header));
                   }
                   return map.entrySet().iterator();
             }

             @Override
             public void put(String key, String value) {
                  this.delegate.addHeader(key, value);
             }
       }
}
```

你可以这样注册:

```
@Bean HttpSpanInjector
customHttpServletResponseSpanInjector() {
```

```
		return new CustomHttpServletResponseSpanInjector();
}

@Bean
HttpResponseInjectingTraceFilter
responseInjectingTraceFilter(Tracer tracer) {
		return new HttpResponseInjectingTraceFilter(tracer,
customHttpServletResponseSpanInjector());
}
```

Zipkin 中的自定义 SA 标签

有时你想创建一个手动 Span，将一个电话包裹到一个没有被检测的外部服务。

您可以做的是创建一个带有 `peer.service` 标签的跨度，其中包含要调用的服务的值。下面你可以看到一个调用 Redis 的例子，它被包装在这样一个跨度里。

```
org.springframework.cloud.sleuth.Span newSpan =
tracer.createSpan("redis");
try {
		newSpan.tag("redis.op", "get");
		newSpan.tag("lc", "redis");
		newSpan.logEvent(org.springframework.cloud.sleuth.S
pan.CLIENT_SEND);
		// call redis service e.g
		// return (SomeObj)
redisTemplate.opsForHash().get("MYHASH", someObjKey);
} finally {
		newSpan.tag("peer.service", "redisService");
		newSpan.tag("peer.ipv4", "1.2.3.4");
		newSpan.tag("peer.port", "1234");
		newSpan.logEvent(org.springframework.cloud.sleuth.S
pan.CLIENT_RECV);
		tracer.close(newSpan);
}
```

重要	记住不要添加 `peer.service` 标签和 `SA` 标签！您只需添加 `peer.service`。

自定义服务名称

默认情况下，Sleuth 假设当您将跨度发送到 Zipkin 时，您希望跨度的服务名称等于 `spring.application.name` 值。这并不总是这样。在某些情况下，您希望为您应用程序中的所有 spans 提供不同的服务名称。要实现这一点，只需将以下属性传递给应用程序即可覆盖该值（`foo` 服务名称的示例）：

```
spring.zipkin.service.name: foo
```

主机定位器

为了定义与特定跨度对应的主机，我们需要解析主机名和端口。默认方法是从服务器属性中获取它。如果由于某些原因没有设置，那么我们正在尝试从网络接口检索主机名。

如果您启用了发现客户端，并且更愿意从服务注册表中注册的实例检索主机地址，那么您必须设置属性（适用于基于 HTTP 和 Stream 的跨度报告）。

```
spring.zipkin.locator.discovery.enabled: true
```

Span Data 作为消息

您可以通过将 `spring-cloud-sleuth-stream` jar 作为依赖关系来累加并发送跨越 Spring Cloud Stream 的数据，并为 RabbitMQ 或 `spring-cloud-starter-stream-kafka` 添加通道 Binder 实现（例如 `spring-cloud-starter-stream-rabbit`）为 Kafka）。通过将 `spring-cloud-sleuth-stream` jar 作为依赖关系，并添加 RabbitMQ 或 `spring-cloud-starter-stream-kafka` 的 Binder 通道 `spring-cloud-starter-stream-rabbit` 来

实现{ 22 /} Stream 的累积和发送范围数据。 Kafka）。这将自动将您的应用程序转换为有效载荷类型为 `Spans` 的邮件的制作者。

Zipkin 消费者

有一个特殊的便利注释，用于为 Span 数据设置消息使用者，并将其推入 Zipkin `SpanStore`。这个应用程序

```
@SpringBootApplication
@EnableZipkinStreamServer
public class Consumer {
	public static void main(String[] args) {
		SpringApplication.run(Consumer.class, args);
	}
}
```

将通过 Spring Cloud Stream`Binder`（例如 RabbitMQ 包含 `spring-cloud-starter-stream-rabbit`）来收听您提供的任何运输的 Span 数据，Redis 和 Kafka 的类似起始者）。如果添加以下 UI 依赖关系

```
<groupId>io.zipkin.java</groupId>
<artifactId>zipkin-autoconfigure-ui</artifactId>
```

然后，您将有一个 Zipkin 服务器，您的应用程序 在端口 9411 上承载 UI 和 API。

默认 `SpanStore` 是内存中的（适合演示，快速入门）。对于更强大的解决方案，您可以将 MySQL 和 `spring-boot-starter-jdbc` 添加到类路径中，并通过配置启用 JDBC `SpanStore`，例如：

```
spring:
```

```
  rabbitmq:
    host: ${RABBIT_HOST:localhost}
  datasource:
    schema: classpath:/mysql.sql
    url: jdbc:mysql://${MYSQL_HOST:localhost}/test
    username: root
    password: root
# Switch this on to create the schema on startup:
    initialize: true
    continueOnError: true
  sleuth:
    enabled: false
zipkin:
  storage:
    type: mysql
```

注意 `@EnableZipkinStreamServer` 还用`@EnableZipkinServer` 注释，因此该过程以通过 HTTP 收集 spans，并在 Zipkin Web UI 中进行查询。

定制消费者

也可以使用 `spring-cloud-sleuth-stream` 并绑定到 `SleuthSink` 来轻松实现自定义消费者。例:

```
@EnableBinding(SleuthSink.class)
@SpringBootApplication(exclude =
SleuthStreamAutoConfiguration.class)
@MessageEndpoint
public class Consumer {

    @ServiceActivator(inputChannel = SleuthSink.INPUT)
    public void sink(Spans input) throws Exception {
        // ... process spans
    }
}
```

注意 上面的示例消费者应用程序明确排除 `SleuthStreamAutoConfiguration`，因此的（您可能实际上想要将消息跟踪到消费者应用程序中）。

为了自定义轮询机制，您可以创建名称等于 `StreamSpanReporter.POLLER` 的 `PollerMetadata` 类型的 bean。在这里可以找到这样一个配置的例子。

```
@Configuration
public static class CustomPollerConfiguration {

	@Bean(name = StreamSpanReporter.POLLER)
	PollerMetadata customPoller() {
		PollerMetadata poller = new
PollerMetadata();
		poller.setMaxMessagesPerPoll(500);
		poller.setTrigger(new
PeriodicTrigger(5000L));
		return poller;
	}
}
```

度量

目前 Spring Cloud Sleuth 注册了与 spans 相关的简单指标。它使用 Spring Boot 的指标支持来计算接受和删除的数量 spans。每次发送到 Zipkin 时，接受的 spans 的数量将增加。如果出现错误，那么删除的数字 spans 将会增加。

集成

可运行和可调用

如果你在 `Runnable` 或 `Callable` 中包含你的逻辑，就可以将这些类包装在他们的 Sleuth 代表中。

`Runnable`的示例:

```
Runnable runnable = new Runnable() {
	@Override
	public void run() {
		// do some work
	}

	@Override
	public String toString() {
		return "spanNameFromToStringMethod";
	}
};
// Manual `TraceRunnable` creation with explicit
"calculateTax" Span name
Runnable traceRunnable = new TraceRunnable(tracer,
spanNamer, runnable, "calculateTax");
// Wrapping `Runnable` with `Tracer`. The Span name will
be taken either from the
// `@SpanName` annotation or from `toString` method
Runnable traceRunnableFromTracer = tracer.wrap(runnable);
```

`Callable`的示例:

```
Callable<String> callable = new Callable<String>() {
	@Override
	public String call() throws Exception {
		return someLogic();
	}

	@Override
	public String toString() {
		return "spanNameFromToStringMethod";
	}
};
// Manual `TraceCallable` creation with explicit
"calculateTax" Span name
Callable<String> traceCallable = new
TraceCallable<>(tracer, spanNamer, callable,
"calculateTax");
// Wrapping `Callable` with `Tracer`. The Span name will
be taken either from the
// `@SpanName` annotation or from `toString` method
```

```
Callable<String> traceCallableFromTracer =
tracer.wrap(callable);
```

这样，您将确保为每次执行创建并关闭新的 Span。

Hystrix

自定义并发策略

我们正在注册 [HystrixConcurrencyStrategy]() 将所有 `Callable` 实例包装到他们的 Sleuth 代表 - `TraceCallable` 中的定制。该策略是启动或继续跨越，这取决于调用 Hystrix 命令之前跟踪是否已经进行的事实。要禁用自定义 Hystrix 并发策略，将 `spring.sleuth.hystrix.strategy.enabled` 设置为 `false`。

手动命令设置

假设您有以下 `HystrixCommand`：

```
HystrixCommand<String> hystrixCommand = new
HystrixCommand<String>(setter) {
	@Override
	protected String run() throws Exception {
		return someLogic();
	}
};
```

为了传递跟踪信息，您必须在 `HystrixCommand` 的 `HystrixCommand` 的 Sleuth 版本中包装相同的逻辑：

```
TraceCommand<String> traceCommand = new
TraceCommand<String>(tracer, traceKeys, setter) {
	@Override
```

```
	public String doRun() throws Exception {
		return someLogic();
	}
};
```

RxJava

我们正在注册 `RxJavaSchedulersHook` 将所有 `Action0` 实例包装到他们的 Sleuth 代表 - `TraceAction` 中的定制。钩子起动或继续一个跨度取决于跟踪在 Action 被安排之前是否已经进行的事实。要禁用自定义 RxJavaSchedulersHook，将 `spring.sleuth.rxjava.schedulers.hook.enabled` 设置为 `false`。

您可以定义线程名称的正则表达式列表，您不希望创建一个 Span。只需在 `spring.sleuth.rxjava.schedulers.ignoredthreads` 属性中提供逗号分隔的正则表达式列表。

HTTP 集成

可以通过提供值等于 `false` 的 `spring.sleuth.web.enabled` 属性来禁用此部分的功能。

HTTP 过滤器

通过 `TraceFilter` 所有采样的进入请求导致创建 Span。Span 的名称是 `http:` +发送请求的路径。例如，如果请求已发送到 `/foo/bar`，则该名称将为 `http:/foo/bar`。您可以通过 `spring.sleuth.web.skipPattern` 属性配置

要跳过的 URI。如果您在类路径上有 `ManagementServerProperties`，则其值 `contextPath` 将附加到提供的跳过模式。

的 HandlerInterceptor

由于我们希望跨度名称是精确的，我们使用的 `TraceHandlerInterceptor` 包装现有的 `HandlerInterceptor`，或直接添加到现有的 `HandlerInterceptors` 列表中。`TraceHandlerInterceptor` 向给定的 `HttpServletRequest` 添加了一个特殊请求属性。如果 `TraceFilter` 没有看到此属性集，它将创建一个“后备”跨度，这是在服务器端创建的一个额外的跨度，以便在 UI 中正确显示跟踪。看到最有可能意味着有一个缺失的仪器。在这种情况下，请在 Spring Cloud Sleuth 中提出问题。

异步 Servlet 支持

如果您的控制器返回 `Callable` 或 `WebAsyncTask` Spring Cloud，Sleuth 将继续现有的跨度，而不是创建一个新的跨度。

HTTP 客户端集成

同步休息模板

我们注入一个 `RestTemplate` 拦截器，确保所有跟踪信息都传递给请求。每次呼叫都会创建一个新的 Span。收到回应后关闭。为了阻止将

`spring.sleuth.web.client.enabled` 设置为 `false` 的同步

`RestTemplate` 功能。

重要	你必须注册 `RestTemplate` 作为一个 bean，以便拦截器被注入。如果您使用 `new` 关 该工具将不工作。

异步休息模板

重要	一个 `AsyncRestTemplate` bean 的跟踪版本是为您开箱即用的。如果你有自己的 `TraceAsyncRestTemplate` 表示来包装它。最好的解决方案是只定制 `ClientHtt` `AsyncClientHttpRequestFactory`。 **如果您有自己的** `AsyncRestTemplate` **踪**。

定制仪器设置为在发送和接收请求时创建和关闭跨度。您可以通过注册您的

bean 来自定义 `ClientHttpRequestFactory` 和

`AsyncClientHttpRequestFactory`。记住使用跟踪兼容的实现（例如，不要

忘记在 `TraceAsyncListenableTaskExecutor` 中包装

`ThreadPoolTaskScheduler`）。自定义请求工厂示例：

```
@EnableAutoConfiguration
@Configuration
public static class TestConfiguration {

	@Bean
	ClientHttpRequestFactory mySyncClientFactory() {
		return new MySyncClientHttpRequestFactory();
	}

	@Bean
	AsyncClientHttpRequestFactory
myAsyncClientFactory() {
		return new
MyAsyncClientHttpRequestFactory();
	}
}
```

将`AsyncRestTemplate`功能集 `spring.sleuth.web.async.client.enabled`阻止为`false`。禁用`TraceAsyncClientHttpRequestFactoryWrapper`设置 `spring.sleuth.web.async.client.factory.enabled`设置为`false`。如果您不想将所有 `spring.sleuth.web.async.client.template.enabled` `false`的`AsyncRestClient`创建为`false`。

Feign

默认情况下，Spring Cloud Sleuth 通过 `TraceFeignClientAutoConfiguration`提供与 feign 的集成。您可以通过将 `spring.sleuth.feign.enabled`设置为 false 来完全禁用它。如果这样做，那么不会发生 Feign 相关的仪器。

Feign 仪器的一部分是通过`FeignBeanPostProcessor`完成的。您可以通过提供`spring.sleuth.feign.processor.enabled`等于`false`来禁用它。如果你这样设置，那么 Spring Cloud Sleuth 不会调整你的任何自定义 Feign 组件。然而，所有默认的工具仍然存在。

异步通信

@Async 注释方法

在 Spring Cloud Sleuth 中，我们正在调用异步相关组件，以便跟踪信息在线程之间传递。您可以通过将 `spring.sleuth.async.enabled` 的值设置为 `false` 来禁用此行为。

如果您使用`@Async`注释方法，那么我们将自动创建一个具有以下特征的新的 Span：

- Span 名称将是注释方法名称
- Span 将被该方法的类名称和方法名称标记

@Scheduled 注释方法

在 Spring Cloud Sleuth 中，我们正在调试计划的方法执行，以便跟踪信息在线程之间传递。您可以通过将 `spring.sleuth.scheduled.enabled` 的值设置为 `false` 来禁用此行为。

如果您使用`@Scheduled`注释方法，那么我们将自动创建一个具有以下特征的新的 Span：

- Span 名称将是注释方法名称
- Span 将被该方法的类名称和方法名称标记

如果要跳过某些`@Scheduled`注释类的 Span 创建，您可以使用与`@Scheduled`注释类的完全限定名称匹配的正则表达式来设置 `spring.sleuth.scheduled.skipPattern`。

提示	如果您一起使用 `spring-cloud-sleuth-stream` 和 `spring-cloud-netflix-` 并发送到 Zipkin。这可能是恼人的。您可以设置 `spring.sleuth.scheduled.skipPattern=org.springframework.cloud.`

Executor，ExecutorService 和 ScheduledExecutorService

我们提供 `LazyTraceExecutor`，`TraceableExecutorService` 和 `TraceableScheduledExecutorService`。每次提交，调用或调度新任务时，这些实现都将创建 Spans。

在这里，您可以看到使用 `CompletableFuture` 使用 `TraceableExecutorService` 传递跟踪信息的示例：

```
CompletableFuture<Long> completableFuture =
CompletableFuture.supplyAsync(() -> {
	// perform some logic
	return 1_000_000L;
}, new TraceableExecutorService(executorService,
		// 'calculateTax' explicitly names the span
- this param is optional
		tracer, traceKeys, spanNamer,
"calculateTax"));
```

消息

Spring Cloud Sleuth 与 Spring Integration 集成。它创建 spans 发布和订阅事件。

要禁用 Spring Integration 检测，请将 `spring.sleuth.integration.enabled` 设置为 false。

您可以提供 `spring.sleuth.integration.patterns` 模式，以明确提供要包括的用于跟踪的通道的名称。默认情况下，所有通道都包含在内。

重要	当使用 `Executor` 构建 Spring Integration `IntegrationFlow` 时，请记住使用 `TraceableExecutorService` 装饰 Spring Integration 执行者频道将导致 spans

Zuul

我们正在注册 Zuul 过滤器来传播跟踪信息（请求标头丰富了跟踪数据）。要禁用 Zuul 支持，请将 `spring.sleuth.zuul.enabled` 属性设置为 `false`。

运行示例

您可以在 Pivotal Web Services 中找到部署的运行示例。在以下链接中查看它们:

- Zipkin 表示样品中的应用程序到顶部
- Zipkin 为啤酒厂在 PWS，其 Github 代码

Spring Cloud Consul

Dalston.RELEASE

该项目通过自动配置并绑定到 Spring 环境和其他 Spring 编程模型成语，为 Spring Boot 应用程序提供 Consul 集成。通过几个简单的注释，您可以快速启用和配置应用程序中的常见模式，并使用基于 Consul 的组件构建大型分布式系

统。提供的模式包括服务发现，控制总线和配置。智能路由（Zuul）和客户端负载平衡（Ribbon），断路器（Hystrix）通过与 Spring Cloud Netflix 的集成提供。

安装 Consul

请参阅安装文档获取有关如何安装 Consul 指令。

Consul Agent

所有 Spring Cloud Consul 应用程序必须可以使用 Consul Agent 客户端。默认情况下，代理客户端预计位于 `localhost:8500`。有关如何启动代理客户端以及如何连接到 Consul Agent 服务器集群的详细信息，请参阅代理文档。对于开发，安装领事后，您可以使用以下命令启动 Consul Agent：

```
./src/main/bash/local_run_consul.sh
```

这将启动端口 8500 上的服务器模式的代理，其中 ui 可以在 http：//localhost：8500 上找到

服务发现与 Consul

服务发现是基于微服务架构的关键原则之一。尝试配置每个客户端或某种形式的约定可能非常困难，可以非常脆弱。Consul 通过 HTTP API 和 DNS 提供服务发现

服务。Spring Cloud Consul 利用 HTTP API 进行服务注册和发现。这不会阻止非 Spring Cloud 应用程序利用 DNS 界面。Consul 代理服务器在通过八卦协议进行通信的集群中运行，并使用 Raft 协议协议。

如何激活

要激活 Consul 服务发现，请使用组 `org.springframework.cloud` 和 artifact id `spring-cloud-starter-consul-discovery` 的启动器。有关使用当前的 Spring Cloud 发布列表设置构建系统的详细信息，请参阅 Spring Cloud 项目页面。

注册 Consul

当客户端注册 Consul 时，它提供有关自身的元数据，如主机和端口，ID，名称和标签。默认情况下会创建一个 HTTP 检查，每隔 10 秒，Consul 命中 `/health` 端点。如果健康检查失败，则服务实例被标记为关键。

示例 Consul 客户端：

```
@SpringBootApplication
@EnableDiscoveryClient
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello world";
    }

    public static void main(String[] args) {
```

```
        new 
SpringApplicationBuilder(Application.class).web(true).run
(args);
    }

}
```

（即完全正常的 Spring Boot 应用程序）。如果 Consul 客户端位于 `localhost:8500` 以外的位置，则需要配置来定位客户端。例：

application.yml

```
spring:
  cloud:
    consul:
      host: localhost
      port: 8500
```

警告	如果您使用 [Spring Cloud Consul Config](), 上述值将需要放在 `bootstrap.yml` 而

来自 `Environment` 的默认服务名称，实例 ID 和端口分别为 `${spring.application.name}`，Spring 上下文 ID 和`${server.port}`。

`@EnableDiscoveryClient` 将应用程序设为 Consul“服务”（即注册自己）和“客户端”（即可以查询 Consul 查找其他服务）。

HTTP 健康检查

Consul 实例的运行状况检查默认为“/ health”，这是 Spring Boot 执行器应用程序中有用端点的默认位置。如果您使用非默认上下文路径或 servlet 路径（例如 `server.servletPath=/foo`）或管理端点路径（例如 `management.context-path=/admin`），则需要更改这些，即使是执行器应

用程序。也可以配置 Consul 用于检查运行状况端点的间隔。“10s”和“1m”分别表示 10 秒和 1 分钟。例：

application.yml

```
spring:
  cloud:
    consul:
      discovery:
        healthCheckPath: ${management.context-path}/health
        healthCheckInterval: 15s
```

元数据和 Consul 标签

Consul 尚未支持服务元数据。Spring Cloud 的 `ServiceInstance` 有一个 `Map<String, String> metadata` 字段。Spring Cloud Consul 使用 Consul 标签来近似元数据，直到 Consul 正式支持元数据。使用 `key=value` 形式的标签将被分割并分别用作 `Map` 键和值。标签没有相同的`=`符号，将被用作键和值两者。

application.yml

```
spring:
  cloud:
    consul:
      discovery:
        tags: foo=bar, baz
```

上述配置将导致具有 `foo→bar` 和 `baz→baz` 的映射。

使 Consul 实例 ID 唯一

默认情况下，一个领事实体注册了一个等于其 Spring 应用程序上下文 ID 的 ID。默认情况下，Spring 应用程序上下文 ID 为 `${spring.application.name}:comma,separated,profiles:${server.port}`。在大多数情况下，这将允许一个服务的多个实例在一台机器上运行。

如果需要进一步的唯一性，使用 Spring Cloud，您可以通过在 `spring.cloud.consul.discovery.instanceId` 中提供唯一的标识来覆盖此。例如：

application.yml

```
spring:
  cloud:
    consul:
      discovery:
        instanceId:
${spring.application.name}:${vcap.application.instance_id
:${spring.application.instance_id:${random.value}}}
```

使用这个元数据和在 localhost 上部署的多个服务实例，随机值将在那里进行，以使实例是唯一的。在 Cloudfoundry 中，`vcap.application.instance_id` 将在 Spring Boot 应用程序中自动填充，因此不需要随机值。

使用 DiscoveryClient

Spring Cloud 支持 Feign（REST 客户端构建器），Spring `RestTemplate` 使用逻辑服务名称而不是物理 URL。

您还可以使用 `org.springframework.cloud.client.discovery.DiscoveryClient`，它为 Netflix 不特定的发现客户端提供了一个简单的 API，例如

```
@Autowired
private DiscoveryClient discoveryClient;

public String serviceUrl() {
    List<ServiceInstance> list =
discoveryClient.getInstances("STORES");
```

```
    if (list != null && list.size() > 0 ) {
        return list.get(0).getUri();
    }
    return null;
}
```

具有 Consul 的分布式配置

Consul 提供了一个用于存储配置和其他元数据的键/值存储。Spring Cloud Consul Config 是 Config Server 和 Client 的替代方案。在特殊的“引导”阶段，配置被加载到 Spring 环境中。默认情况下，配置存储在`/config`文件夹中。基于应用程序的名称和模拟解析属性的 Spring Cloud Config 顺序的活动配置文件创建多个`PropertySource`实例。例如，名为“testApp”的应用程序和“dev”配置文件将创建以下属性源：

```
config/testApp,dev/
config/testApp/
config/application,dev/
config/application/
```

最具体的物业来源位于顶部，底部最不具体。Properties 是`config/application`文件夹适用于使用 consul 进行配置的所有应用程序。`config/testApp`文件夹中的 Properties 仅适用于名为“testApp”的服务实例。

配置当前在应用程序启动时被读取。发送 HTTP POST 到`/refresh`将导致配置被重新加载。观看关键价值商店（Consul 支持））目前不可能，但将来将是此项目的补充。

如何激活

要开始使用 Consul 配置，请使用组 `org.springframework.cloud` 和 artifact id `spring-cloud-starter-consul-config` 的启动器。有关使用当前的 Spring Cloud 发布列表设置构建系统的详细信息，请参阅 Spring Cloud 项目页面。

这将启用自动配置，将设置 Spring Cloud Consul 配置。

定制

Consul 可以使用以下属性定制配置：

bootstrap.yml

```
spring:
  cloud:
    consul:
      config:
        enabled: true
        prefix: configuration
        defaultContext: apps
        profileSeparator: '::'
```

- `enabled` 将此值设置为“false”禁用 Consul 配置
- `prefix` 设置配置值的基本文件夹
- `defaultContext` 设置所有应用程序使用的文件夹名称
- `profileSeparator` 设置用于使用配置文件在属性源中分隔配置文件名称的分隔符的值

配置观察

Consul 配置观察功能可以利用领事看守钥匙前缀的能力。Config Watch 会阻止 Consul HTTP API 调用，以确定当前应用程序是否有任何相关配置数据发生更改。如果有新的配置数据，则会发布刷新事件。这相当于调用`/refresh`执行器端点。

要更改 Config Watch 调用的频率

change `spring.cloud.consul.config.watch.delay`。默认值为 1000，以毫秒为单位。

禁用配置观察集 `spring.cloud.consul.config.watch.enabled=false`。

YAML 或 Properties 配置

与单个键/值对相反，可以更方便地将 YBL 或 Properties 格式的属性块存储起来。将 `spring.cloud.consul.config.format` 属性设置为 `YAML` 或 `PROPERTIES`。例如使用 YAML：

bootstrap.yml

```
spring:
  cloud:
    consul:
      config:
        format: YAML
```

YAML 必须在合适的 `data` 键中设置。使用键上面的默认值将如下所示：

```
config/testApp,dev/data
```

```
config/testApp/data
config/application,dev/data
config/application/data
```

您可以将 YAML 文档存储在上述任何键中。

您可以使用`spring.cloud.consul.config.data-key`更改数据密钥。

git2consul 与配置

git2consul 是一个 Consul 社区项目，将文件从 git 存储库加载到各个密钥到 Consul。默认情况下，密钥的名称是文件的名称。YAML 和 Properties 文件分别支持`.yml`和`.properties`的文件扩展名。将`spring.cloud.consul.config.format`属性设置为`FILES`。例如：

bootstrap.yml

```
spring:
  cloud:
    consul:
      config:
        format: FILES
```

给定`/config`中的以下密钥，`development`配置文件和应用程序名称为`foo`：

```
.gitignore
application.yml
bar.properties
foo-development.properties
foo-production.yml
foo.properties
master.ref
```

将创建以下属性来源：

```
config/foo-development.properties
```

```
config/foo.properties
config/application.yml
```

每个键的值需要是一个格式正确的 YAML 或 Properties 文件。

快速失败

在某些情况下（如本地开发或某些测试场景）可能会方便，如果不能配置领事，则不会失败。在 `bootstrap.yml` 中设置 `spring.cloud.consul.config.failFast=false` 将导致配置模块记录一个警告而不是抛出异常。这将允许应用程序继续正常启动。

Consul 重试

如果您希望您的应用程序启动时可能偶尔无法使用代理商，则可以要求您在发生故障后继续尝试。您需要在您的类路径中添加 `spring-retry` 和 `spring-boot-starter-aop`。默认行为是重试 6 次，初始退避间隔为 1000ms，指数乘数为 1.1，用于后续退避。您可以使用 `spring.cloud.consul.retry.*`配置属性配置这些属性（和其他）。这适用于 Spring Cloud Consul 配置和发现注册。

提示	要完全控制重试，请使用 id 为“consulRetryInterceptor”添加 `RetryOperatio` Spring 重试有一个 `RetryInterceptorBuilder` 可以轻松创建一个。

Spring Cloud Bus 与 Consul

如何激活

要开始使用 Consul 总线，使用组 `org.springframework.cloud` 和工件 id `spring-cloud-starter-consul-bus` 的起动器。有关使用当前的 Spring Cloud 发布列表设置构建系统的详细信息，请参阅 Spring Cloud 项目页面。

有关可用的执行机构端点以及如何发送自定义消息，请参阅 Spring Cloud Bus 文档。

断路器与 Hystrix

应用程序可以使用 Spring Cloud Netflix 项目提供的 Hystrix 断路器将这个启动器包含在项目 pom.xml：`spring-cloud-starter-hystrix` 中。Hystrix 不依赖于 Netflix Discovery Client。`@EnableHystrix` 注释应放置在配置类（通常是主类）上。那么方法可以用`@HystrixCommand` 注释来被断路器保护。有关详细信息，请参阅文档。

使用 Turbine 和 Consul Hystrix 指标聚合

Turbine（由 Spring Cloud Netflix 项目提供））聚合多个实例 Hystrix 指标流，因此仪表板可以显示聚合视图。Turbine 使用 `DiscoveryClient` 接口查找相关实

例。要将 Turbine 与 Spring Cloud Consul 结合使用，请按以下示例配置 Turbine 应用程序：

的 pom.xml

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-netflix-turbine</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-
discovery</artifactId>
</dependency>
```

请注意，Turbine 依赖不是起始者。涡轮启动器包括对 Netflix Eureka 的支持。

application.yml

```
spring.application.name: turbine
applications: consulhystrixclient
turbine:
  aggregator:
    clusterConfig: ${applications}
  appConfig: ${applications}
```

`clusterConfig` 和 `appConfig` 部分必须匹配，因此将逗号分隔的服务标识列表放在单独的配置属性中是有用的。

Turbine。java 的

```
@EnableTurbine
@EnableDiscoveryClient
@SpringBootApplication
public class Turbine {
    public static void main(String[] args) {

SpringApplication.run(DemoturbinecommonsApplication.class
, args);
    }
}
```

Spring Cloud Zookeeper

该项目通过自动配置并绑定到 Spring 环境和其他 Spring 编程模型成语，为 Spring Boot 应用程序提供 Zookeeper 集成。通过几个简单的注释，您可以快速启用和配置应用程序中的常见模式，并使用基于 Zookeeper 的组件构建大型分布式系统。提供的模式包括服务发现和配置。智能路由（Zuul）和客户端负载平衡（Ribbon），断路器（Hystrix）通过与 Spring Cloud Netflix 的集成提供。

安装 Zookeeper

请参阅安装文档获取有关如何安装 Zookeeper 指令。

服务发现与 Zookeeper

服务发现是基于微服务架构的关键原则之一。尝试配置每个客户端或某种形式的约定可能非常困难，可以非常脆弱。策展人（一个用于 Zookeeper 的 java 库）通过服务发现扩展提供服务发现服务。Spring Cloud Zookeeper 利用此扩展功能进行服务注册和发现。

如何激活

包括对 `org.springframework.cloud:spring-cloud-starter-zookeeper-discovery` 的依赖将启用将设置 Spring Cloud Zookeeper 发现的自动配置。

注意	您仍然需要包含 `org.springframework.boot:spring-boot-starter-web` 的

注册 Zookeeper

当客户端注册 Zookeeper 时，它提供有关自身的元数据，如主机和端口，ID 和名称。

示例 Zookeeper 客户端：

```
@SpringBootApplication
@EnableDiscoveryClient
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello world";
    }

    public static void main(String[] args) {
        new
SpringApplicationBuilder(Application.class).web(true).run
(args);
    }

}
```

（即完全正常的 Spring Boot 应用程序）。如果 Zookeeper 位于 `localhost:2181` 以外的地方，则需要配置来定位服务器。例：

application.yml

```
spring:
  cloud:
    zookeeper:
      connect-string: localhost:2181
```

警告	如果您使用 [Spring Cloud Zookeeper 配置]，上述值将需要放置在 `bootstrap.yml`

来自 `Environment` 的默认服务名称，实例 ID 和端口分别为

`${spring.application.name}`，Spring 上下文 ID 和`${server.port}`。

`@EnableDiscoveryClient` 将应用程序同时进入 Zookeeper“服务”（即注册自己）和“客户端”（即可以查询 Zookeeper 查找其他服务）。

使用 DiscoveryClient

Spring Cloud 支持 Feign（REST 客户端构建器），还支持 Spring `RestTemplate` 使用逻辑服务名称而不是物理 URL。

您还可以使用

`org.springframework.cloud.client.discovery.DiscoveryClient`，

它为 Netflix 不具体的发现客户端提供简单的 API，例如

```
@Autowired
private DiscoveryClient discoveryClient;

public String serviceUrl() {
    List<ServiceInstance> list =
discoveryClient.getInstances("STORES");
    if (list != null && list.size() > 0 ) {
        return list.get(0).getUri().toString();
    }
    return null;
}
```

使用 Spring Cloud Zookeeper 与 Spring Cloud Netflix 组件

Spring Cloud Netflix 提供有用的工具，无论使用哪种 `DiscoveryClient` 实现。Feign，Turbine，Ribbon 和 Zuul 均与 Spring Cloud Zookeeper 合作。

Ribbon 与 Zookeeper

Spring Cloud Zookeeper 提供 Ribbon 的 `ServerList` 的实现。当使用 `spring-cloud-starter-zookeeper-discovery` 时，Ribbon 默认情况下自动配置为使用 `ZookeeperServerList`。

Spring Cloud Zookeeper 和服务注册表

Spring Cloud Zookeeper 实现 `ServiceRegistry` 接口，允许开发人员以编程方式注册任意服务。

`ServiceInstanceRegistration` 类提供 `builder()` 方法来创建可以由 `ServiceRegistry` 使用的 `Registration` 对象。

```
@Autowired
```

```
private ZookeeperServiceRegistry serviceRegistry;

public void registerThings() {
    ZookeeperRegistration registration =
ServiceInstanceRegistration.builder()
            .defaultUriSpec()
            .address("anyUrl")
            .port(10)
            .name("/a/b/c/d/anotherservice")
            .build();
    this.serviceRegistry.register(registration);
}
```

实例状态

Netflix Eureka 支持在服务器上注册的实例是 `OUT_OF_SERVICE`，而不是作为活动服务实例返回。这对于诸如蓝色/绿色部署之类的行为非常有用。策展人服务发现配方不支持此行为。利用灵活的有效载荷，让 Spring Cloud Zookeeper 通过更新一些特定的元数据，然后对 Ribbon `ZookeeperServerList` 中的元数据进行过滤来实现 `OUT_OF_SERVICE`。`ZookeeperServerList` 过滤出不等于 `UP` 的所有非空实例状态。如果实例状态字段为空，则向后兼容性被认为是 `UP`。将实例 POST `OUT_OF_SERVICE` 的状态更改为 `ServiceRegistry` 实例状态执行器端点。

```
----
$ echo -n OUT_OF_SERVICE | http POST
http://localhost:8081/service-registry/instance-status
----
NOTE: The above example uses the `http` command from
https://httpie.org
```

Zookeeper 依赖关系

使用 Zookeeper 依赖关系

Spring Cloud Zookeeper 可以让您提供应用程序的依赖关系作为属性。作为依赖关系，您可以了解 Zookeeper 中注册的其他应用程序，您可以通过 [Feign]（REST 客户端构建器）以及 [Spring `RestTemplate`] 呼叫。

您还可以从 Zookeeper 依赖关系观察者功能中受益，这些功能可让您控制和监视依赖关系的状态，并决定如何处理。

如何激活 Zookeeper 依赖关系

- 包括对 `org.springframework.cloud:spring-cloud-starter-zookeeper-discovery` 的依赖将启用将自动配置 Spring Cloud Zookeeper 依赖关系的自动配置。
- 如果您必须正确设置 `spring.cloud.zookeeper.dependencies` 部分 - 请查看后续部分以获取更多详细信息，然后该功能处于活动状态
- 即使您在属性中提供依赖关系，也可以关闭依赖关系。只需将属性 `spring.cloud.zookeeper.dependency.enabled` 设置为 false（默认为 `true`）。

设置 Zookeeper 依赖关系

我们来仔细看一下依赖关系表示的例子：

application.yml

```
spring.application.name: yourServiceName
spring.cloud.zookeeper:
  dependencies:
    newsletter:
      path:
/path/where/newsletter/has/registered/in/zookeeper
      loadBalancerType: ROUND_ROBIN
      contentTypeTemplate:
application/vnd.newsletter.$version+json
      version: v1
      headers:
        header1:
            - value1
        header2:
            - value2
      required: false
      stubs: org.springframework:foo:stubs
    mailing:
      path:
/path/where/mailing/has/registered/in/zookeeper
      loadBalancerType: ROUND_ROBIN
      contentTypeTemplate:
application/vnd.mailing.$version+json
      version: v1
      required: true
```

现在让我们一个接一个地遍历依赖的每个部分。根属性名称为 `spring.cloud.zookeeper.dependencies`。

别名

在根属性下面，由于 Ribbon 的限制，必须通过别名来表示每个依赖关系（应用程序 ID 必须放在 URL 中，因此您不能传递任何复杂的路径，如/ foo / bar / name ）。别名将是您将使用的名称，而不是 `DiscoveryClient`，`Feign` 或 `RestTemplate` 的 serviceId。

在上述例子中，别名是 `newsletter` 和 `mailing`。使用 `newsletter` 的 Feign 使用示例为：

```
@FeignClient("newsletter")
public interface NewsletterService {
        @RequestMapping(method = RequestMethod.GET, value =
"/newsletter")
        String getNewsletters();
}
```

路径

代表 `path` yaml 属性。

Path 是根据 Zookeeper 注册依赖关系的路径。像 Ribbon 之前提交的 URL，因此这个路径不符合其要求。这就是为什么 Spring Cloud Zookeeper 将别名映射到正确的路径。

负载平衡器类型

代表 `loadBalancerType` yaml 属性。

如果您知道在调用此特定依赖关系时必须应用什么样的负载平衡策略，那么您可以在 yaml 文件中提供它，并将自动应用。您可以选择以下负载平衡策略之一

- STICKY - 一旦选择了该实例将始终被调用
- 随机 - 随机选择一个实例
- ROUND_ROBIN - 一遍又一遍地迭代实例

Content-Type 模板和版本

代表 `contentTypeTemplate` 和 `version` yaml 属性。

如果您通过 `Content-Type` 标题版本您的 api，那么您不想将此标头添加到您的每个请求中。另外如果你想调用一个新版本的 API，你不想漫游你的代码，以增加 API 版本。这就是为什么您可以提供 `contentTypeTemplate` 特殊 `$version` 占位符的原因。该占位符将由 `version` yaml 属性的值填充。我们来看一个例子。

拥有以下 `contentTypeTemplate`：

```
application/vnd.newsletter.$version+json
```

和以下 `version`：

```
v1
```

将导致为每个请求设置 `Content-Type` 标题：

```
application/vnd.newsletter.v1+json
```

默认标题

由 yaml 代表 `headers` 映射

有时每次调用依赖关系都需要设置一些默认标头。为了不在代码中这样做，您可以在 yaml 文件中设置它们。拥有以下 `headers` 部分：

```
headers:
    Accept:
        - text/html
```

```
        - application/xhtml+xml
    Cache-Control:
        - no-cache
```

结果在您的 HTTP 请求中添加适当的值列表的 `Accept` 和 `Cache-Control` 标头。

强制依赖

在 yaml 中由 `required` 属性表示

如果您的一个依赖关系在您的应用程序启动时需要启动并运行，则可以在 yaml 文件中设置 `required: true` 属性。

如果您的应用程序无法在引导期间本地化所需的依赖关系，则会抛出异常，并且 Spring 上下文将无法设置。换句话说，如果 Zookeeper 中没有注册所需的依赖关系，则您的应用程序将无法启动。

您可以在以下部分阅读有关 Spring Cloud Zookeeper 存在检查器的更多信息。

存根

您可以为包含依赖关系的存根的 JAR 提供冒号分隔路径。例

```
stubs: org.springframework:foo:stubs
```

意味着对于特定的依赖关系可以在下面找到：

- groupId：`org.springframework`
- artifactId：`foo`

- 分类器：`stubs` - 这是默认值

这实际上等于

```
stubs: org.springframework:foo
```

因为 `stubs` 是默认分类器。

配置 Spring Cloud Zookeeper 依赖关系

有一些属性可以设置为启用/禁用 Zookeeper 依赖关系功能的部分。

- `spring.cloud.zookeeper.dependencies` - 如果您不设置此属性，则不会从 Zookeeper 依赖关系中受益
- `spring.cloud.zookeeper.dependency.ribbon.enabled`（默认情况下启用） - Ribbon 需要显式的全局配置或特定的依赖关系。通过打开此属性，运行时负载平衡策略解决是可能的，您可以从 Zookeeper 依赖关系的 `loadBalancerType` 部分获利。需要此属性的配置具有 `LoadBalancerClient` 的实现，委托给下一个子弹中的 `ILoadBalancer`
- `spring.cloud.zookeeper.dependency.ribbon.loadbalancer`（默认情况下启用） - 感谢这个属性，自定义 `ILoadBalancer` 知道传递给 Ribbon 的 URI 部分实际上可能是必须被解析为 Zookeeper。没有此属性，您将无法在嵌套路径下注册应用程序。
- `spring.cloud.zookeeper.dependency.headers.enabled`（默认情况下启用） - 此属性注册这样的一个 `RibbonClient`，它会自动附加适当的头

文件和内容类型，其中包含依赖关系配置中显示的版本。没有这两个参数的设置将不会运行。

- `spring.cloud.zookeeper.dependency.resttemplate.enabled`（默认情况下启用） - 启用时将修改`@LoadBalanced`注释的 `RestTemplate` 的请求标头，以便它通过依赖关系配置中设置的版本的标题和内容类型。Wihtout 这两个参数的设置将无法运行。

Spring Cloud Zookeeper 依赖关系观察者

依赖关系观察器机制允许您将侦听器注册到依赖关系中。功能实际上是 `Observator` 模式的实现。当依赖关系改变其状态（UP 或 DOWN）时，可以应用一些自定义逻辑。

如何激活

Spring Cloud Zookeeper 依赖关系功能需要启用从依赖关系观察器机制中获利。

注册听众

为了注册一个监听器，你必须实现一个接口 `org.springframework.cloud.zookeeper.discovery.watcher.Depend`

`encyWatcherListener`，并将其注册为一个 bean。该界面为您提供了一种方法：

```
   void stateChanged(String dependencyName,
DependencyState newState);
```

如果要为特定的依赖关系注册一个侦听器，那么 `dependencyName` 将是具体实现的鉴别器。`newState` 将提供您的依赖关系是否已更改为 `CONNECTED` 或 `DISCONNECTED` 的信息。

存在检查

绑定与依赖关系观察器是称为存在检查器的功能。它允许您在启动应用程序时提供自定义行为，以根据您的依赖关系的状态作出反应。

抽象 `org.springframework.cloud.zookeeper.discovery.watcher.presence.DependencyPresenceOnStartupVerifier` 类的默认实现是 `org.springframework.cloud.zookeeper.discovery.watcher.presence.DefaultDependencyPresenceOnStartupVerifier`，它以以下方式工作。

- 如果依赖关系标记为我们 `required`，并且不在 Zookeeper 中，则在引导时，您的应用程序将抛出异常并关闭
- 如果依赖关系不是 `required`，`org.springframework.cloud.zookeeper.discovery.watcher.pre`

`sence.LogMissingDependencyChecker`将在`WARN`级别上记录该应用程序

功能可以被覆盖，因为只有当没有`DependencyPresenceOnStartupVerifier`的 bean 时才会注册`DefaultDependencyPresenceOnStartupVerifier`。

分布式配置与 Zookeeper

Zookeeper 提供了一个分层命名空间，允许客户端存储任意数据，如配置数据。Spring Cloud Zookeeper Config 是 Config Server 和 Client 的替代方案。在特殊的"引导"阶段，配置被加载到 Spring 环境中。默认情况下，配置存储在`/config`命名空间中。根据应用程序的名称和模拟解析属性的 Spring Cloud Config 顺序的活动配置文件，创建多个`PropertySource`实例。例如，名为"testApp"的应用程序和"dev"配置文件将创建以下属性源：

```
config/testApp,dev
config/testApp
config/application,dev
config/application
```

最具体的物业来源位于顶部，底部最不具体。Properties 是`config/application`命名空间适用于使用 zookeeper 进行配置的所有应用程序。`config/testApp`命名空间中的 Properties 仅适用于名为"testApp"的服务实例。

配置当前在应用程序启动时被读取。发送 HTTP POST 到`/refresh`将导致重新加载配置。观看配置命名空间（Zookeeper 支持））目前尚未实现，但将来将会添加到此项目中。

如何激活

包括对`org.springframework.cloud:spring-cloud-starter-zookeeper-config`的依赖将启用将配置 Spring Cloud Zookeeper 配置的自动配置。

定制

Zookeeper 可以使用以下属性自定义配置：

bootstrap.yml

```
spring:
  cloud:
    zookeeper:
      config:
        enabled: true
        root: configuration
        defaultContext: apps
        profileSeparator: '::'
```

- `enabled`将此值设置为“false”将禁用 Zookeeper 配置
- `root` 设置配置值的基本命名空间
- `defaultContext` 设置所有应用程序使用的名称
- `profileSeparator` 设置用于使用配置文件在属性源中分隔配置文件名称的分隔符的值

spring-cloud.adoc 中的未解决的指令 - include :: / Users / sgibb / workspace / spring / spring-cloud-samples / scripts / docs /../ cli / docs / src / main / asciidoc / spring-cloud-cli。 ADOC []

Spring Cloud Security

Spring Cloud Security 提供了一组用于构建安全应用程序和服务的原语，最小化。可以从外部（或集中）高度配置的声明式模型适用于通常使用中央契约管理服务的大型合作远程组件系统的实现。在像 Cloud Foundry 这样的服务平台上也很容易使用。基于 Spring Boot 和 Spring 安全性 OAuth2，我们可以快速创建实现常见模式的系统，如单点登录，令牌中继和令牌交换。

注意	Spring Cloud 根据非限制性 Apache 2.0 许可证发布。如果您想为文档的这一部分做 中找到项目中的源代码和问题跟踪器。

快速开始

OAuth2 单一登录

这是一个具有 HTTP 基本身份验证和单个用户帐户的 Spring Cloud“Hello World”应用程序

app.groovy

```
@Grab('spring-boot-starter-security')
@Controller
class Application {

  @RequestMapping('/')
```

```
  String home() {
    'Hello World'
  }

}
```

您可以使用 `spring run app.groovy` 运行它，并观察日志的密码（用户名为"用户"）。到目前为止，这只是一个 Spring Boot 应用程序的默认设置。

这是一个使用 OAuth2 SSO 的 Spring Cloud 应用程序:

app.groovy

```
@Controller
@EnableOAuth2Sso
class Application {

  @RequestMapping('/')
  String home() {
    'Hello World'
  }

}
```

指出不同？这个应用程序的行为实际上与之前的一样，因为它不知道它是 OAuth2 的信誉。

您可以很容易地在 github 注册一个应用程序，所以如果你想要一个生产应用程序在你自己的域上尝试。如果您很乐意在 localhost：8080 上测试，那么请在应用程序配置中设置这些属性:

application.yml

```
security:
  oauth2:
    client:
      clientId: bd1c0a783ccdd1c9b9e4
      clientSecret:
1a9030fbca47a5b2c28e92f19050bb77824b5ad1
```

```
      accessTokenUri:
https://github.com/login/oauth/access_token
      userAuthorizationUri:
https://github.com/login/oauth/authorize
      clientAuthenticationScheme: form
    resource:
      userInfoUri: https://api.github.com/user
      preferTokenInfo: false
```

运行上面的应用程序，它将重定向到 github 进行授权。如果您已经登录 github，您甚至不会注意到它已经通过身份验证。只有您的应用程序在 8080 端口上运行，这些凭据才会起作用。

要限制客户端在获取访问令牌时要求的范围，您可以设置 `security.oauth2.client.scope`（逗号分隔或 YAML 中的数组）。默认情况下，作用域为空，由授权服务器确定默认值是什么，通常取决于客户端注册中的设置。

注意	上面的例子都是 Groovy 脚本。如果要在 Java（或 Groovy）中编写相同的代码，则 类路径中（例如，请参阅此处的 示例）。

OAuth2 受保护资源

您要使用 OAuth2 令牌保护 API 资源？这是一个简单的例子（与上面的客户端配对）：

app.groovy

```
@Grab('spring-cloud-starter-security')
@RestController
@EnableResourceServer
class Application {

  @RequestMapping('/')
```

```
  def home() {
    [message: 'Hello World']
  }

}
```

和

application.yml

```
security:
  oauth2:
    resource:
      userInfoUri: https://api.github.com/user
      preferTokenInfo: false
```

更多详情

单点登录

注意	所有 OAuth2 SSO 和资源服务器功能在版本 1.3 中移动到 Spring Boot。您可以在 Sp

令牌中继

令牌中继是 OAuth2 消费者充当客户端，并将传入令牌转发到外发资源请求。消费者可以是纯客户端（如 SSO 应用程序）或资源服务器。

客户端令牌中继

如果您的应用是面向 OAuth2 客户端的用户（即声明为`@EnableOAuth2Sso`或`@EnableOAuth2Client`），那么它的请求范围为 spring security 的`OAuth2ClientContext`。您可以从此上下文和自动连线

OAuth2ProtectedResourceDetails创建自己的 OAuth2RestTemplate，然后上下文将始终向下转发访问令牌，如果过期则自动刷新访问令牌。（这些是 Spring 安全和 Spring Boot 的功能。）

注意	如果您使用 client_credentials 令牌，则 Spring Boot（1.4.1）不会自动创建 OAuth2ProtectedResourceDetails。在这种情况下，您需要创建自己的 Clie 并使用@ConfigurationProperties("security.oauth2.client")进行配置

客户端令牌中继在 Zuul 代理

如果您的应用程式还有 Spring Cloud Zuul 嵌入式反向代理（使用 @EnableZuulProxy），那么您可以要求将 OAuth2 访问令牌转发到其正在代理的服务。因此，上述的 SSO 应用程序可以简单地增强：

app.groovy

```
@Controller
@EnableOAuth2Sso
@EnableZuulProxy
class Application {

}
```

并且（除了将用户登录并抓取令牌之外）将下载的身份验证令牌传递到 /proxy/*服务。如果这些服务是用@EnableResourceServer 实现的，那么他们将在正确的标题中获得一个有效的标记。

它是如何工作的？@EnableOAuth2Sso 注释引入 spring-cloud-starter-security（您可以在传统应用程序中手动执行），而这又会触发一个 ZuulFilter 的自动配置，该属性本身被激活，因为 Zuul 在 classpath（通过

`@EnableZuulProxy`）。该过滤器仅从当前已认证的用户提取访问令牌，并将其放入下游请求的请求头中。

资源服务器令牌中继

如果您的应用有`@EnableResourceServer`，您可能希望将传入令牌下载到其他服务。如果您使用`RestTemplate`联系下游服务，那么这只是如何使用正确的上下文创建模板的问题。

如果您的服务使用`UserInfoTokenServices`验证传入令牌（即正在使用`security.oauth2.user-info-uri`配置）），则可以使用自动连线`OAuth2ClientContext`创建`OAuth2RestTemplate`（将由身份验证过程之前它遇到后端代码）。相等（使用 Spring Boot 1.4），您可以在配置中注入`UserInfoRestTemplateFactory`并抓取其中的`OAuth2RestTemplate`。例如：

MyConfiguration.java

```
@Bean
public OAuth2RestTemplate
restTemplate(UserInfoRestTemplateFactory factory) {
    return factory.getUserInfoRestTemplate();
}
```

然后，此休息模板将具有由身份验证过滤器使用的`OAuth2ClientContext`（请求作用域）相同，因此您可以使用它来发送具有相同访问令牌的请求。

如果您的应用没有使用`UserInfoTokenServices`，但仍然是客户端（即声明`@EnableOAuth2Client`或`@EnableOAuth2Sso`），则使用 Spring 安全云任何

`OAuth2RestOperations`，用户从`@Autowired` `@OAuth2Context` 也会转发令牌。此功能默认实现为 MVC 处理程序拦截器，因此它仅适用于 Spring MVC。如果您不使用 MVC，可以使用包含 `AccessTokenContextRelay` 的自定义过滤器或 AOP 拦截器来提供相同的功能。

以下是一个基本示例，显示了使用其他地方创建的自动连线休息模板（“foo.com”是一个资源服务器，接受与周围应用程序相同的令牌）：

MyController.java

```
@Autowired
private OAuth2RestOperations restTemplate;

@RequestMapping("/relay")
public String relay() {
    ResponseEntity<String> response =
      restTemplate.getForEntity("https://foo.com/bar",
String.class);
    return "Success! (" + response.getBody() + ")";
}
```

如果您不想转发令牌（这是一个有效的选择，因为您可能希望以自己的身份而不是向您发送令牌的客户端），那么您只需要创建自己的 `OAuth2Context` 的自动装配默认值。

Feign 客户端也会选择使用 `OAuth2ClientContext` 的拦截器，如果它是可用的，那么他们还应该在 `RestTemplate` 将要执行的令牌中继。

配置 Zuul 代理下游的认证

您可以通过`proxy.auth.*`设置控制`@EnableZuulProxy`下游的授权行为。

例:

application.yml
```
proxy:
  auth:
    routes:
      customers: oauth2
      stores: passthru
      recommendations: none
```

在此示例中，“客户”服务获取 OAuth2 令牌中继，“存储”服务获取传递（授权头只是通过下游），“建议”服务已删除其授权头。如果有令牌可用，则默认行为是执行令牌中继，否则为 passthru。

有关详细信息，请参阅 ProxyAuthenticationProperties。

Spring Cloud 为 Cloud Foundry

Cloudfoundry 的 Spring Cloud 可以轻松地在 Cloud Foundry（平台即服务）中运行 Spring Cloud 应用程序。Cloud Foundry 有一个“服务”的概念，它是“绑定”到应用程序的中间件，本质上为其提供包含凭据的环境变量（例如，用于服务的位置和用户名）。

`spring-cloud-cloudfoundry-web` 项目为 Cloud Foundry 中的 webapps 的一些增强功能提供基本支持：自动绑定到单点登录服务，并可选择启用粘性路由进行发现。

`spring-cloud-cloudfoundry-discovery` 项目提供 Spring Cloud Commons `DiscoveryClient` 的实施，因此您可以 `@EnableDiscoveryClient` 并将您的凭据提供为 `spring.cloud.cloudfoundry.discovery.[email,password]`，然后直接或通过 `LoadBalancerClient` 使用 `DiscoveryClient` /}（如果您没有连接到 Pivotal Web Services，则也为`*.url`）。

第一次使用它时，发现客户端可能很慢，因为它必须从 Cloud Foundry 获取访问令牌。

发现

以下是 Cloud Foundry 发现的 Spring Cloud 应用程序：

app.groovy

```
@Grab('org.springframework.cloud:spring-cloud-
cloudfoundry')
@RestController
@EnableDiscoveryClient
class Application {

  @Autowired
  DiscoveryClient client

  @RequestMapping('/')
  String home() {
    'Hello from ' + client.getLocalServiceInstance()
  }

}
```

如果您运行它没有任何服务绑定：

```
$ spring jar app.jar app.groovy
$ cf push -p app.jar
```

它将在主页中显示其应用程序名称。

`DiscoveryClient`可以根据身份验证的凭据列出空间中的所有应用程序，其中的空间默认为客户端运行的空间（如果有的话）。如果组织和空间都不配置，则它们将根据 Cloud Foundry 中的用户配置文件进行默认。

单点登录

注意	所有 OAuth2 SSO 和资源服务器功能在版本 1.3 中移动到 Spring Boot。您可以在 Sp

该项目提供从 CloudFoundry 服务凭据到 Spring Boot 功能的自动绑定。如果您有一个称为“sso”的 CloudFoundry 服务，例如，使用包含“client_id”，“client_secret”和“auth_domain”的凭据，它将自动绑定到您使用`@EnableOAuth2Sso`启用的 Spring OAuth2 客户端来自 Spring Boot）。可以使用`spring.oauth2.sso.serviceId`对服务的名称进行参数化。

Spring Cloud Contract

文献作者：Adam Dudczak，MathiasDüsterhöft，Marcin Grzejszczak，Dennis Kieselhorst，JakubKubryński，Karol Lassak，Olga Maciaszek-Sharma，MariuszSmykuła，Dave Syer

Dalston.RELEASE

Spring Cloud Contract

您始终需要的是将新功能推向分布式系统中的新应用程序或服务的信心。该项目为 Spring 应用程序中的消费者驱动 Contracts 和服务架构提供支持，涵盖了一系列用于编写测试的选项，将其作为资产发布，声称生产者和消费者保留合同用于 HTTP 和消息的交互。

Spring Cloud Contract WireMock

模块让您有可能使用 WireMock 使用嵌入在 Spring Boot 应用的“环境”服务器不同的服务器。查看 样品 了解更多详情。

重要	Spring Cloud 发布列表 BOM 导入 `spring-cloud-contract-dependencies`，这关系。这可能导致一种情况，即使你不使用 Spring Cloud Contract，那么你的依赖

如果您有一个使用 Tomcat 作为嵌入式服务器的 Spring Boot 应用程序（默认为 `spring-boot-starter-web`））， 那么您可以简单地将 `spring-cloud-contract-wiremock` 添加到类路径中并添加 `@AutoConfigureWireMock`，以便可以在测试中使用 Wiremock。Wiremock 作为存根服务器运行，您可以使用 Java API 或通过静态 JSON 声明来注册存根行为，作为测试的一部分。这是一个简单的例子：

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment =
WebEnvironment.RANDOM_PORT)
@AutoConfigureWireMock(port = 0)
public class WiremockForDocsTests {
      // A service that calls out over HTTP
```

```
        @Autowired private Service service;

        // Using the WireMock APIs in the normal way:
        @Test
        public void contextLoads() throws Exception {
                // Stubbing WireMock
                stubFor(get(urlEqualTo("/resource"))

        .willReturn(aResponse().withHeader("Content-Type",
"text/plain").withBody("Hello World!")));
                // We're asserting if WireMock responded
properly

        assertThat(this.service.go()).isEqualTo("Hello
World!");
        }

}
```

要使用`@AutoConfigureWireMock(port=9999)`（例如）启动不同端口上的存根服务器，并且对于随机端口使用值 0.存根服务器端口将在测试应用程序上下文中绑定为“wiremock.server.port”。使用`@AutoConfigureWireMock`将一个类型为`WiremockConfiguration`的 bean 添加到测试应用程序上下文中，它将被缓存在具有相同上下文的方法和类之间，就像一般的 Spring 集成测试一样。

自动注册存根

如果您使用`@AutoConfigureWireMock`，则它将从文件系统或类路径注册 WireMock JSON 存根，默认情况下为 `file:src/test/resources/mappings`。您可以使用注释中的`stubs`属性自定义位置，这可以是资源模式（ant-style）或目录，在这种情况下，附加`*/.json`。例:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureWireMock(stubs="classpath:/stubs")
public class WiremockImportApplicationTests {

	@Autowired
	private Service service;

	@Test
	public void contextLoads() throws Exception {

	assertThat(this.service.go()).isEqualTo("Hello
World!");
	}

}
```

注意	实际上，WireMock 总是从 `src/test/resources/mappings` 中加载映射以及行为，您还必须如下所述指定文件根。

使用文件指定存根体

WireMock 可以从类路径或文件系统上的文件读取响应体。在这种情况下，您将在 JSON DSL 中看到响应具有“bodyFileName”而不是（文字）“body”。默认情况下，相对于根目录 `src/test/resources/__files` 解析文件。要自定义此位置，您可以将`@AutoConfigureWireMock` 注释中的 `files` 属性设置为父目录的位置（即，位置`__files` 是子目录）。您可以使用 Spring 资源符号来引用 `file:…`或 `classpath:…`位置（但不支持通用 URL）。可以给出值列表，并且 WireMock 将在需要查找响应体时解析存在的第一个文件。

注意	当配置 `files` 根时，它会影响自动加载存根（它们来自称为“映射”的子目录中的性明确加载的存根没有影响。

替代方法：使用 JUnit 规则

对于更常规的 WireMock 体验，使用 JUnit `@Rules` 启动和停止服务器，只需使用 `WireMockSpring` 便利类来获取 `Options` 实例：

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment =
WebEnvironment.RANDOM_PORT)
public class WiremockForDocsClassRuleTests {

	// Start WireMock on some dynamic port
	// for some reason `dynamicPort()` is not working
properly
	@ClassRule
	public static WireMockClassRule wiremock = new
WireMockClassRule(

	WireMockSpring.options().dynamicPort());
	// A service that calls out over HTTP to
localhost:${wiremock.port}
	@Autowired
	private Service service;

	// Using the WireMock APIs in the normal way:
	@Test
	public void contextLoads() throws Exception {
		// Stubbing WireMock

	wiremock.stubFor(get(urlEqualTo("/resource"))

	.willReturn(aResponse().withHeader("Content-Type",
"text/plain").withBody("Hello World!")));
		// We're asserting if WireMock responded
properly

	assertThat(this.service.go()).isEqualTo("Hello
World!");
	}

}
```

使用`@ClassRule`表示服务器将在此类中的所有方法后关闭。

WireMock 和 Spring MVC 模拟器

Spring Cloud Contract 提供了一个方便的类，可以将 JSON WireMock 存根加载到 Spring `MockRestServiceServer` 中。以下是一个例子：

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
public class WiremockForDocsMockServerApplicationTests {

	@Autowired
	private RestTemplate restTemplate;

	@Autowired
	private Service service;

	@Test
	public void contextLoads() throws Exception {
		// will read stubs classpath
		MockRestServiceServer server =
WireMockRestServiceServer.with(this.restTemplate)

	.baseUrl("http://example.org").stubs("classpath:/st
ubs/resource.json")
				.build();
		// We're asserting if WireMock responded
properly

	assertThat(this.service.go()).isEqualTo("Hello
World");
		server.verify();
	}
}
```

`baseUrl`前面是所有模拟调用，`stubs()`方法将一个存根路径资源模式作为参数。所以在这个例子中，`/stubs/resource.json`定义的存根被加载到模拟服

务器中，所以如果 `RestTemplate` 被要求访问 http://example.org/，那么它将得到所声明的响应。可以指定多个存根模式，每个可以是一个目录（对于所有“.json”的递归列表）或一个固定的文件名（如上例所示）或一个蚂蚁样式模式。JSON 格式是通常的 WireMock 格式，您可以在 WireMock 网站上阅读。

目前，我们支持 Tomcat，Jetty 和 Undertow 作为 Spring Boot 嵌入式服务器，而 Wiremock 本身对特定版本的 Jetty（目前为 9.2）具有“本机”支持。要使用本地 Jetty，您需要添加本机线程依赖关系，并排除 Spring Boot 容器（如果有的话）。

使用 RestDocs 生成存根

Spring RestDocs 可用于为具有 Spring MockMvc 或 RestEasy 的 HTTP API 生成文档（例如，asciidoctor 格式）。在生成 API 文档的同时，还可以使用 Spring Cloud Contract WireMock 生成 WireMock 存根。只需编写正常的 RestDocs 测试用例，并使用`@AutoConfigureRestDocs` 在 restdocs 输出目录中自动存储存根。例如：

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureRestDocs(outputDir = "target/snippets")
@AutoConfigureMockMvc
public class ApplicationTests {

	@Autowired
	private MockMvc mockMvc;

	@Test
	public void contextLoads() throws Exception {
```

```
            mockMvc.perform(get("/resource"))

      .andExpect(content().string("Hello World"))
                        .andDo(document("resource"));
      }
}
```

从此测试将在"target / snippets / stubs / resource.json"上生成一个 WireMock 存根。它将所有 GET 请求与"/ resource"路径相匹配。

没有任何其他配置，这将创建一个存根与 HTTP 方法的请求匹配器和除"主机"和"内容长度"之外的所有头。为了更准确地匹配请求，例如要匹配 POST 或 PUT 的正文，我们需要明确创建一个请求匹配器。这将做两件事情：1）创建一个只匹配您指定的方式的存根，2）断言测试用例中的请求也匹配相同的条件。

主要的入口点是`WireMockRestDocs.verify()`，可以替代`document()`便利方法。例如：

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureRestDocs(outputDir = "target/snippets")
@AutoConfigureMockMvc
public class ApplicationTests {

      @Autowired
      private MockMvc mockMvc;

      @Test
      public void contextLoads() throws Exception {
            mockMvc.perform(post("/resource")
             .content("{\"id\":\"123456\",\"message\":\"H
ello World\"}"))
                        .andExpect(status().isOk())
                        .andDo(verify().jsonPath("$.id")
                  .stub("resource"));
      }
}
```

所以这个合同是说：任何有效的 POST 与“id”字段将得到与本测试相同的响应。

您可以将来电链接到`.jsonPath()`以添加其他匹配器。如果 您不熟悉 JayWay 文档，可以帮助您加快 JSON 路径的速度。

您也可以使用 WireMock API 来验证请求是否与创建的存根匹配，而不是使用`jsonPath`和`contentType`方法。例：

```
@Test
public void contextLoads() throws Exception {
       mockMvc.perform(post("/resource")
              .content("{\"id\":\"123456\",\"message\":\"He
llo World\"}"))
                     .andExpect(status().isOk())
                     .andDo(verify()
                                 .wiremock(WireMock.post(

       urlPathEquals("/resource"))

       .withRequestBody(matchingJsonPath("$.id"))
                     .stub("post-resource"));
}
```

WireMock API 是丰富的 - 您可以通过正则表达式以及 json 路径来匹配头文件，查询参数和请求正文，因此这可以用于创建具有更广泛参数的存根。上面的例子会生成一个这样的 stub：

后 resource.json

```
{
  "request" : {
    "url" : "/resource",
    "method" : "POST",
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$.id"
    }]
  },
  "response" : {
```

```
    "status" : 200,
    "body" : "Hello World",
    "headers" : {
      "X-Application-Context" : "application:-1",
      "Content-Type" : "text/plain"
    }
  }
}
```

注意	您可以使用 `wiremock()` 方法或 `jsonPath()` 和 `contentType()` 方法创建请求匹

在消费方面，假设上面生成的 `resource.json` 可以在类路径中使用，您可以使用 WireMock 以多种不同的方式创建一个存根，其中包括上述使用 `@AutoConfigureWireMock(stubs="classpath:resource.json")` 的描述。

使用 RestDocs 生成 Contracts

可以使用 Spring RestDocs 生成的另一件事是 Spring Cloud Contract DSL 文件和文档。如果您将其与 Spring Cloud WireMock 相结合，那么您将获得合同和存根。

提示	您可能会想知道为什么该功能在 WireMock 模块中。来想一想，它确实有道理，因为 义。这就是为什么我们建议做这两个。

我们来想象下面的测试：

```
                this.mockMvc.perform(post("/foo")

        .accept(MediaType.APPLICATION_PDF)

        .accept(MediaType.APPLICATION_JSON)
```

```
        .contentType(MediaType.APPLICATION_JSON)
                                .content("{\"foo\":
23 }"))
                        .andExpect(status().isOk())

        .andExpect(content().string("bar"))
                        // first WireMock
                        .andDo(WireMockRestDocs.verify()

        .jsonPath("$[?(@.foo >= 20)]")

        .contentType(MediaType.valueOf("application/json"))

        .stub("shouldGrantABeerIfOldEnough"))
                        // then Contract DSL
documentation
                        .andDo(document("index",
SpringCloudContractRestDocs.dslContract()));
```

这将导致在上一节中介绍的存根的创建，合同将被生成和文档文件。

合同将被称为 `index.groovy`，看起来更像是这样。

```
import org.springframework.cloud.contract.spec.Contract

Contract.make {
    request {
        method 'POST'
        url 'http://localhost:8080/foo'
        body('''
            {"foo": 23 }
        ''')
        headers {
            header('''Accept''', '''application/json''')
            header('''Content-Type''',
'''application/json''')
            header('''Host''', '''localhost:8080''')
            header('''Content-Length''', '''12''')
        }
    }
    response {
        status 200
```

```
        body('''
        bar
        ''')
        headers {
            header('''Content-Type''',
'''application/json;charset=UTF-8''')
            header('''Content-Length''', '''3''')
        }
        testMatchers {
            jsonPath('$[?(@.foo >= 20)]', byType())
        }
    }
}
```

生成的文档（Asciidoc 的示例）将包含格式化的合同（此文件的位置将为 `index/dsl-contract.adoc`）。

Spring Cloud Contract 验证者

介绍

重要	1.1.0 版中已弃用的 Accurest 项目的文档可在此处获取。
提示	Accurest 项目最初是由 Marcin Grzejszczak 和 Jakub Kubrynski（codearte.io）

只是为了简短说明 - Spring Cloud Contract 验证程序是一种能够启用消费者驱动合同（CDC）开发基于 JVM 的应用程序的工具。它与*合同定义语言*（DSL）一起提供。合同定义用于生成以下资源：

- 在客户端代码（*客户端测试*）上进行集成测试时，WireMock 将使用 JSON 存根定义。测试代码仍然需要手动编写，测试数据由 Spring Cloud Contract 验证器生成。

- 消息传递路由，如果你使用一个。我们正在与 Spring Integration，Spring Cloud Stream，Spring AMQP 和 Apache Camel 进行整合。然而，您可以设置自己的集成，如果你想
- 验收测试（在 JUnit 或 Spock 中）用于验证 API 的服务器端实现是否符合合同（*服务器测试*）。完全测试由 Spring Cloud Contract 验证器生成。

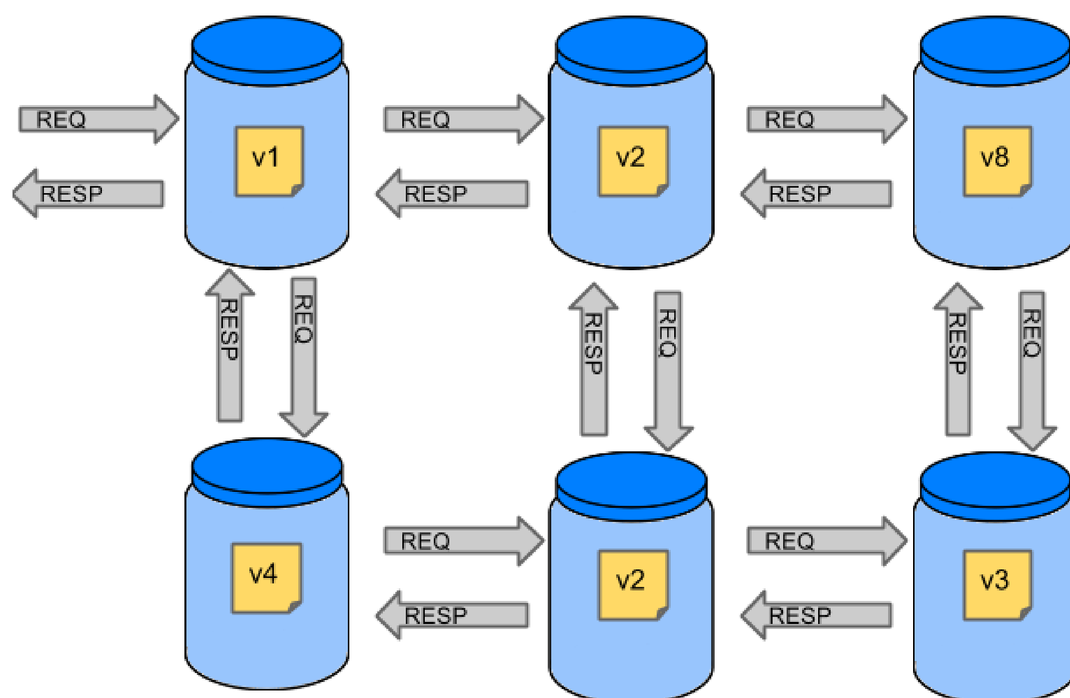
Spring Cloud Contract 验证者将 TDD 移动到软件体系结构的层次。

Spring Cloud Contract 视频

您可以查看华沙 JUG 关于 Spring Cloud Contract 的视频：[点击此处查看视频]

为什么?

让我们假设我们有一个由多个微服务组成的系统：



测试问题

如果我们想测试应用程序在左上角，如果它可以与其他服务通信，那么我们可以做两件事之一：

- 部署所有微服务器并执行端到端测试
- 模拟其他微型服务单元/集成测试

两者都有其优点，但也有很多缺点。我们来关注后者。

部署所有微服务器并执行端到端测试

优点：

- 模拟生产
- 测试服务之间的真实沟通

缺点：

- 要测试一个微服务器，我们将不得不部署 6 个微服务器，几个数据库等。
- 将进行测试的环境将被锁定用于一套测试（即没有人能够在此期间运行测试）。
- 长跑
- 非常迟的反馈
- 非常难调试

模拟其他微型服务单元/集成测试

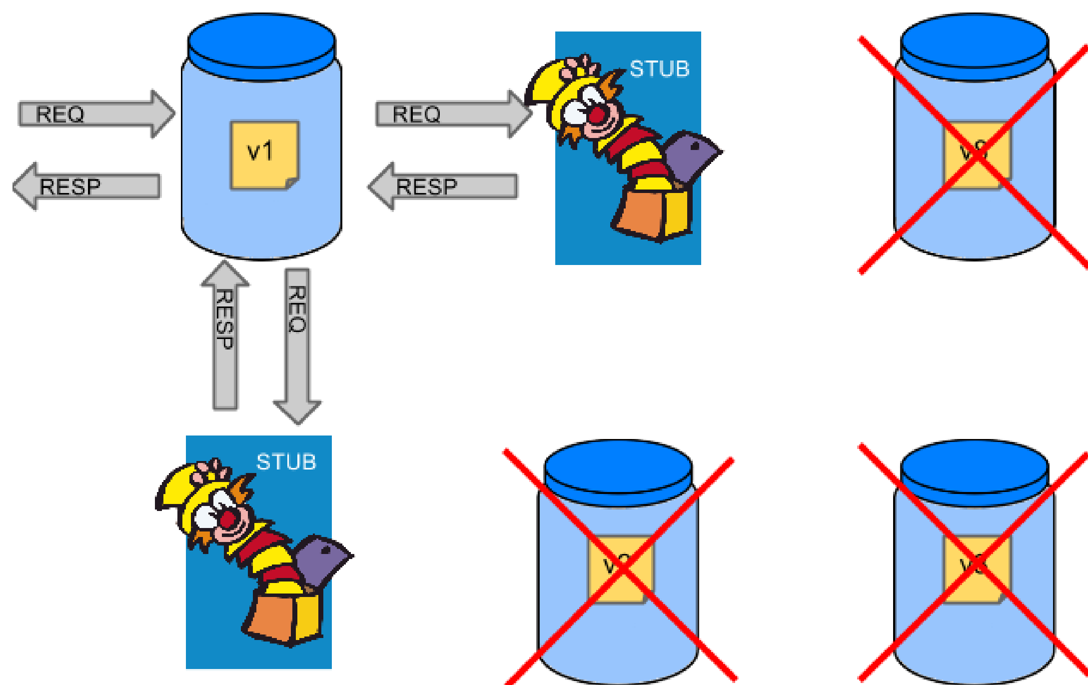
优点：

- 非常快的反馈
- 没有基础架构要求

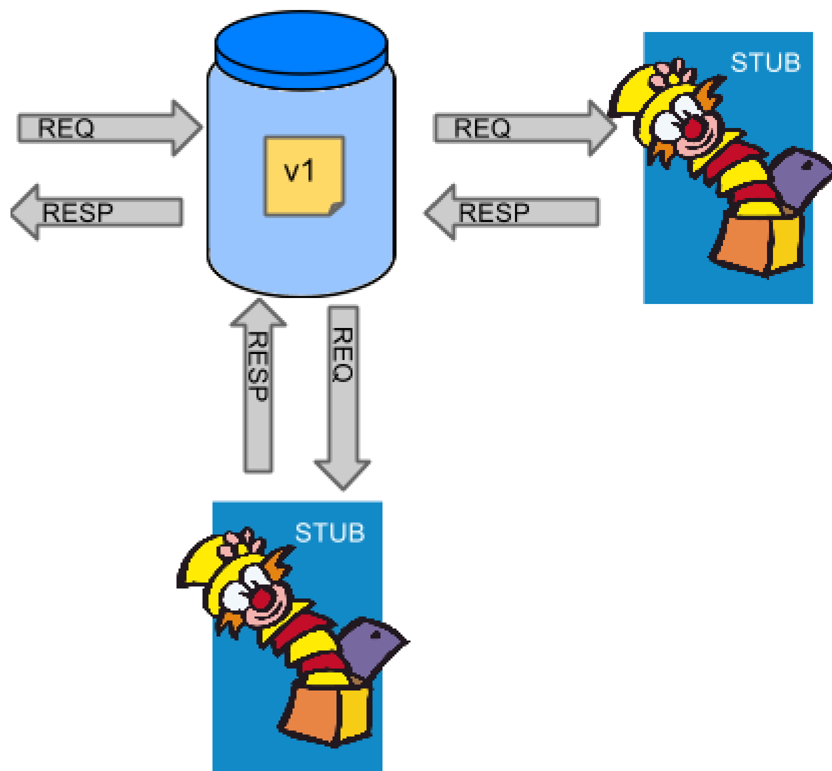
缺点：

- 服务的实现者创建存根，因此它们可能与现实无关
- 您可以通过测试和生产不合格进行生产

为了解决上述问题，Spring Cloud Contract Stub Runner 的验证器被创建。他们的主要思想是给你非常快的反馈，而不需要建立整个微服务的世界。



如果您在存根上工作，那么您需要的唯一应用是应用程序直接使用的应用程序。



Spring Cloud Contract 验证者确定您使用的存根是由您正在调用的服务创建的。此外，如果您可以使用它们，这意味着它们是针对生产者的一方进行测试的。换句话说 - 你可以信任这些存根。

目的

Spring Cloud Contract 验证器与 Stub Runner 的主要目的是：

- 确保 WireMock / Messaging 存根（在开发客户端时使用）正在完全实际执行服务器端实现，
- 推广 ATDD 方法和微服务架构风格，
- 提供一种发布双方立即可见的合同变更的方法，
- 生成服务器端使用的样板测试代码。

重要	Spring Cloud Contract 验证者的目的不是开始在合同中编写业务功能。我们假设我一个用户因为 100 个不同的原因而被欺骗，我们假设你会创建 2 个合同。一个为正面测试应用程序之间的合同，而不是模拟完整行为。

客户端

在测试期间，您希望启动并运行一个模拟服务 Y 的 WireMock 实例/消息传递路由。您希望为该实例提供适当的存根定义。该存根定义将需要有效，并且也应在服务器端可重用。

总结: 在这方面，在存根定义中，您可以使用模式进行请求存根，并需要确切的响应值。

服务器端

作为开发您的存根的服务 Y，您需要确保它实际上类似于您的具体实现。您不能以某种方式存在您的存根行为，并且您的生产应用程序以不同的方式运行。

这就是为什么会生成提供的存根验收测试，这将确保您的应用程序的行为与您在存根中定义的相同。

总结: 在这方面，在存根定义中，您需要精确的值作为请求，并可以使用模式/方法进行响应验证。

逐步向 CDC 指导

举一个欺诈检测和贷款发行流程的例子。业务情景是这样的，我们想向人们发放贷款，但不希望他们从我们那里偷钱。目前我们的系统实施给大家贷款。

假设 `Loan Issuance` 是 `Fraud Detection` 服务器的客户端。在目前的冲刺中，我们需要开发一个新的功能 - 如果客户想要借到太多的钱，那么我们将他们标记为欺诈。

技术说明 - 欺诈检测将具有工件 ID `http-server`，贷款发行 `http-client`，两者都具有组 ID `com.example`。

社会声明 - 客户端和服务器开发团队都需要直接沟通，并在整个过程中讨论变更。CDC 是关于沟通的。

在服务器端代码可以在这里和这里的客户端代码。

提示	在这种情况下，合同的所有权在生产者方面。这意味着物理上所有的合同都存在于生

技术说明

如果使用 **SNAPSHOT / 里程碑 / 版本候选**版本，请将以下部分添加到您的

Maven 的

```
<repositories>
	<repository>
		<id>spring-snapshots</id>
		<name>Spring Snapshots</name>
		<url>https://repo.spring.io/snapshot</url>
		<snapshots>
			<enabled>true</enabled>
		</snapshots>
	</repository>
	<repository>
		<id>spring-milestones</id>
		<name>Spring Milestones</name>
		<url>https://repo.spring.io/milestone</url>
		<snapshots>
```

```
                <enabled>false</enabled>
            </snapshots>
        </repository>
        <repository>
            <id>spring-releases</id>
            <name>Spring Releases</name>
            <url>https://repo.spring.io/release</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
</repositories>
<pluginRepositories>
        <pluginRepository>
            <id>spring-snapshots</id>
            <name>Spring Snapshots</name>
            <url>https://repo.spring.io/snapshot</url>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </pluginRepository>
        <pluginRepository>
            <id>spring-milestones</id>
            <name>Spring Milestones</name>
            <url>https://repo.spring.io/milestone</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </pluginRepository>
        <pluginRepository>
            <id>spring-releases</id>
            <name>Spring Releases</name>
            <url>https://repo.spring.io/release</url>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </pluginRepository>
</pluginRepositories>
```

摇篮

```
repositories {
        mavenCentral()
        mavenLocal()
        maven { url "http://repo.spring.io/snapshot" }
```

```
        maven { url "http://repo.spring.io/milestone" }
        maven { url "http://repo.spring.io/release" }
}
```

消费方（贷款发行）

作为贷款发行服务（欺诈检测服务器的消费者）的开发人员：

通过对您的功能进行测试开始做 TDD

```
@Test
public void shouldBeRejectedDueToAbnormalLoanAmount() {
        // given:
        LoanApplication application = new
LoanApplication(new Client("1234567890"),
                        99999);
        // when:
        LoanApplicationResult loanApplication =
service.loanApplication(application);
        // then:
        assertThat(loanApplication.getLoanApplicationStatus
())

        .isEqualTo(LoanApplicationStatus.LOAN_APPLICATION_R
EJECTED);
        assertThat(loanApplication.getRejectionReason()).is
EqualTo("Amount too high");
}
```

我们刚刚写了一个关于我们新功能的测试。如果收到大额的贷款申请，我们应该拒绝有一些描述的贷款申请。

写入缺少的实现

在某些时间点，您需要向欺诈检测服务发送请求。我们假设我们要发送包含客户端 ID 和要从我们借款的金额的请求。我们想通过 `PUT` 方法将其发送到 `/fraudcheck` 网址。

```
ResponseEntity<FraudServiceResponse> response =
             restTemplate.exchange("http://localhost:" +
port + "/fraudcheck", HttpMethod.PUT,
                         new HttpEntity<>(request,
httpHeaders),
                         FraudServiceResponse.class);
```

为了简单起见，我们已将 `8080` 的欺诈检测服务端口硬编码，我们的应用程序正在 `8090` 上运行。

如果我们开始写测试，显然会因为端口 `8080` 上没有运行而打断。

在本地克隆欺诈检测服务存储库

我们将开始玩服务器端的合同。这就是为什么我们需要先克隆它。

```
git clone https://your-git-server.com/server-side.git
local-http-server-repo
```

在欺诈检测服务的回购中本地定义合同

作为消费者，我们需要确定我们想要实现的目标。我们需要制定我们的期望。这就是为什么我们写下面的合同。

重要	我们将合同放在 `src/test/resources/contract/fraud` 下。`fraud` 文件夹是 类中引用该文件夹。

```
package contracts

org.springframework.cloud.contract.spec.Contract.make {
```

```
	request { // (1)
		method 'PUT' // (2)
		url '/fraudcheck' // (3)
		body([ // (4)
			clientId: $(regex('[0-9]{10}')),
			loanAmount: 99999
		])
		headers { // (5)

	contentType('application/vnd.fraud.v1+json')
		}
	}
	response { // (6)
		status 200 // (7)
		body([ // (8)
			fraudCheckStatus: "FRAUD",
			rejectionReason: "Amount too high"
		])
		headers { // (9)

	contentType('application/vnd.fraud.v1+json')
		}
	}
}

/*
Since we don't want to force on the user to hardcode
values of fields that are dynamic
(timestamps, database ids etc.), one can parametrize
those entries. If you wrap your field's
 value in a `$(...)` or `value(...)` and provide a
dynamic value of a field then
 the concrete value will be generated for you. If you
want to be really explicit about
 which side gets which value you can do that by using the
`value(consumer(...), producer(...))` notation.
 That way what's present in the `consumer` section will
end up in the produced stub. What's
 there in the `producer` will end up in the autogenerated
test. If you provide only the
 regular expression side without the concrete value then
Spring Cloud Contract will generate one for you.
```

```
From the Consumer perspective, when shooting a request in
the integration test:

(1) - If the consumer sends a request
(2) - With the "PUT" method
(3) - to the URL "/fraudcheck"
(4) - with the JSON body that
 * has a field `clientId` that matches a regular
expression `[0-9]{10}`
 * has a field `loanAmount` that is equal to `99999`
(5) - with header `Content-Type` equal to
`application/vnd.fraud.v1+json`
(6) - then the response will be sent with
(7) - status equal `200`
(8) - and JSON body equal to
 { "fraudCheckStatus": "FRAUD", "rejectionReason":
"Amount too high" }
(9) - with header `Content-Type` equal to
`application/vnd.fraud.v1+json`

From the Producer perspective, in the autogenerated
producer-side test:

(1) - A request will be sent to the producer
(2) - With the "PUT" method
(3) - to the URL "/fraudcheck"
(4) - with the JSON body that
 * has a field `clientId` that will have a generated
value that matches a regular expression `[0-9]{10}`
 * has a field `loanAmount` that is equal to `99999`
(5) - with header `Content-Type` equal to
`application/vnd.fraud.v1+json`
(6) - then the test will assert if the response has been
sent with
(7) - status equal `200`
(8) - and JSON body equal to
 { "fraudCheckStatus": "FRAUD", "rejectionReason":
"Amount too high" }
(9) - with header `Content-Type` matching
`application/vnd.fraud.v1+json.*`
 */
```

合同使用静态类型的 Groovy DSL 编写。你可能想知道这些 `value(client(…),server(…))` 部分是什么。通过使用这种符号 Spring Cloud Contract，您可以定义动态的 JSON / URL /等的部分。在标识符或时间戳的情况下，您不想硬编码一个值。你想允许一些不同的值范围。这就是为什么对于消费者端，您可以设置与这些值匹配的正则表达式。您可以通过地图符号或带插值的 String 来提供身体。 有关更多信息，请参阅文档。我们强烈推荐使用地图符号！

提示	了解地图符号设置合同非常重要。请阅读有关 JSON 的 Groovy 文档

上述合同是双方达成的协议：

- 如果发送了 HTTP 请求
  - 端点`/fraudcheck` 上的方法 `PUT`
  - 与 `clientPesel` 的正则表达式`[0-9]{10}`和 `loanAmount` 等于 `99999` 的 JSON 体
  - 并且标题 `Content-Type` 等于 `application/vnd.fraud.v1+json`
- 那么 HTTP 响应将被发送给消费者
  - 状态为 `200`
  - 包含 JSON 体，其中包含值为 `FRAUD` 的 `fraudCheckStatus` 字段和值为 `Amount too high` 的 `rejectionReason` 字段
  - 和一个值为 `application/vnd.fraud.v1+json` 的 `Content-Type` 标头

一旦我们准备好在集成测试中实际检查 API，我们需要在本地安装存根

添加 Spring Cloud Contract 验证程序插件

我们可以添加 Maven 或 Gradle 插件 - 在这个例子中，我们将展示如何添加 Maven。首先我们需要添加`Spring Cloud Contract`BOM。

```
<dependencyManagement>
      <dependencies>
            <dependency>

      <groupId>org.springframework.cloud</groupId>
                  <artifactId>spring-cloud-
dependencies</artifactId>
                  <version>${spring-cloud-
dependencies.version}</version>
                  <type>pom</type>
                  <scope>import</scope>
            </dependency>
      </dependencies>
</dependencyManagement>
```

接下来，`Spring Cloud Contract Verifier` Maven 插件

```
<plugin>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-maven-
plugin</artifactId>
      <version>${spring-cloud-contract.version}</version>
      <extensions>true</extensions>
      <configuration>

      <packageWithBaseClasses>com.example.fraud</packageW
ithBaseClasses>
      </configuration>
</plugin>
```

自添加插件后，我们将从提供的合同中获得`Spring Cloud Contract Verifier`功能：

- 生成并运行测试
- 生产并安装存根

我们不想生成测试，因为我们作为消费者，只想玩短线。这就是为什么我们需要跳过测试生成和执行。当我们执行：

```
cd local-http-server-repo
./mvnw clean install -DskipTests
```

在日志中，我们将看到如下：

```
[INFO] --- spring-cloud-contract-maven-
plugin:1.0.0.BUILD-SNAPSHOT:generateStubs (default-
generateStubs) @ http-server ---
[INFO] Building jar: /some/path/http-server/target/http-
server-0.0.1-SNAPSHOT-stubs.jar
[INFO]
[INFO] --- maven-jar-plugin:2.6:jar (default-jar) @ http-
server ---
[INFO] Building jar: /some/path/http-server/target/http-
server-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.0.BUILD-
SNAPSHOT:repackage (default) @ http-server ---
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-
install) @ http-server ---
[INFO] Installing /some/path/http-server/target/http-
server-0.0.1-SNAPSHOT.jar to
/path/to/your/.m2/repository/com/example/http-
server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT.jar
[INFO] Installing /some/path/http-server/pom.xml to
/path/to/your/.m2/repository/com/example/http-
server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT.pom
```

```
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar to
/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
```

这条线是非常重要的


```
[INFO] Installing /some/path/http-server/target/http-server-0.0.1-SNAPSHOT-stubs.jar to
/path/to/your/.m2/repository/com/example/http-server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-stubs.jar
```


确认 `http-server` 的存根已安装在本地存储库中。

运行集成测试

为了从自动存根下载的 Spring Cloud Contract Stub Runner 功能中获利，您必须在我们的消费者端项目（`Loan Application service`）中执行以下操作。

添加 `Spring Cloud Contract` BOM

```
<dependencyManagement>
	<dependencies>
		<dependency>

	<groupId>org.springframework.cloud</groupId>
			<artifactId>spring-cloud-dependencies</artifactId>
			<version>${spring-cloud-dependencies.version}</version>
			<type>pom</type>
			<scope>import</scope>
		</dependency>
	</dependencies>
</dependencyManagement>
```

将依赖关系添加到`Spring Cloud Contract Stub Runner`

```
<dependency>
	<groupId>org.springframework.cloud</groupId>
	<artifactId>spring-cloud-starter-contract-stub-
runner</artifactId>
	<scope>test</scope>
</dependency>
```

用`@AutoConfigureStubRunner`标注你的测试课程。在注释中，提供 Stub Runner 下载协作者存根的组 ID 和工件 ID。离线工作开关还可以离线使用协作者（可选步骤）。

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.NONE)
@AutoConfigureStubRunner(ids = {"com.example:http-server-
dsl:+:stubs:6565"}, workOffline = true)
@DirtiesContext
public class LoanApplicationServiceTests {
```

现在如果你运行测试你会看到这样的:

```
2016-07-19 14:22:25.403  INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader  : Desired version
is + - will try to resolve the latest version
2016-07-19 14:22:25.438  INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader  : Resolved
version is 0.0.1-SNAPSHOT
2016-07-19 14:22:25.439  INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader  : Resolving
artifact com.example:http-server:jar:stubs:0.0.1-SNAPSHOT
using remote repositories []
2016-07-19 14:22:25.451  INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader  : Resolved
artifact com.example:http-server:jar:stubs:0.0.1-SNAPSHOT
to /path/to/your/.m2/repository/com/example/http-
server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-
stubs.jar
2016-07-19 14:22:25.465  INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader  : Unpacking stub
```

```
from JAR [URI:
file:/path/to/your/.m2/repository/com/example/http-
server/0.0.1-SNAPSHOT/http-server-0.0.1-SNAPSHOT-
stubs.jar]
2016-07-19 14:22:25.475  INFO 41050 --- [           main]
o.s.c.c.stubrunner.AetherStubDownloader  : Unpacked file
to
[/var/folders/0p/xwq47sq106x1_g3dtv6qfm940000gq/T/contrac
ts100276532569594265]
2016-07-19 14:22:27.737  INFO 41050 --- [           main]
o.s.c.c.stubrunner.StubRunnerExecutor    : All stubs are
now running RunningStubs
[namesAndPorts={com.example:http-server:0.0.1-
SNAPSHOT:stubs=8080}]
```

这意味着 Stub Runner 找到了您的存根，并为具有组 ID 为 `com.example`，artifact id `http-server`，版本为 `0.0.1-SNAPSHOT` 的存根和 `stubs` 分类器的端口 `8080`。

档案公关

我们到现在为止是一个迭代的过程。我们可以玩合同，安装在本地，在消费者身边工作，直到我们对合同感到满意。

一旦我们对结果感到满意，测试通过将 PR 发布到服务器端。目前消费者方面的工作已经完成。

生产者方（欺诈检测服务器）

作为欺诈检测服务器（贷款发行服务的服务器）的开发人员：

初步实施

作为提醒，您可以看到初始实现

```
@RequestMapping(
            value = "/fraudcheck",
            method = PUT,
            consumes = FRAUD_SERVICE_JSON_VERSION_1,
            produces = FRAUD_SERVICE_JSON_VERSION_1)
public FraudCheckResult fraudCheck(@RequestBody
FraudCheck fraudCheck) {
return new FraudCheckResult(FraudCheckStatus.OK,
NO_REASON);
}
```

接管公关

```
git checkout -b contract-change-pr master
git pull https://your-git-server.com/server-side-fork.git
contract-change-pr
```

您必须添加自动生成测试所需的依赖关系

```
      <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-contract-
verifier</artifactId>
      <scope>test</scope>
</dependency>
```

在 Maven 插件的配置中，我们传递了 `packageWithBaseClasses` 属性

```
<plugin>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-maven-
plugin</artifactId>
      <version>${spring-cloud-contract.version}</version>
      <extensions>true</extensions>
      <configuration>

      <packageWithBaseClasses>com.example.fraud</packageW
ithBaseClasses>
      </configuration>
```

```
</plugin>
```

重要	我们决定使用“约定”命名，方法是设置 `packageWithBaseClasses` 属性。这意类的名称。在我们这个例子中，这些合约是 `src/test/resources/contract/f` 件夹开始有 2 个包，我们只挑选一个是 `fraud`。我们正在添加 `Base` 后缀，我们正在测试类名称。

这是因为所有生成的测试都会扩展该类。在那里你可以设置你的 Spring 上下文

或任何必要的。在我们的例子中，我们使用 Rest Assured MVC 来启动服务器端

`FraudDetectionController`。

```
package com.example.fraud;

import org.junit.Before;

import
com.jayway.restassured.module.mockmvc.RestAssuredMockMvc;

public class FraudBase {
	@Before
	public void setup() {
		RestAssuredMockMvc.standaloneSetup(new
FraudDetectionController(),
				new
FraudStatsController(stubbedStatsProvider()));
	}

	private StatsProvider stubbedStatsProvider() {
		return fraudType -> {
			switch (fraudType) {
			case DRUNKS:
				return 100;
			case ALL:
				return 200;
			}
			return 0;
		};
	}

	public void assertThatRejectionReasonIsNull(Object
rejectionReason) {
```

```
            assert rejectionReason == null;
      }
}
```

现在，如果你运行`./mvnw clean install`，你会得到这样的 sth：

```
Results :

Tests in error:
  ContractVerifierTest.validate_shouldMarkClientAsFraud:32
» IllegalState Parsed...
```

这是因为您有一个新的合同，从中生成测试，并且由于您尚未实现该功能而失败。自动生成测试将如下所示：

```
@Test
public void validate_shouldMarkClientAsFraud() throws
Exception {
    // given:
        MockMvcRequestSpecification request = given()
                .header("Content-Type",
"application/vnd.fraud.v1+json")
                .body("{\"clientPesel\":\"1234567890\",\"loa
nAmount\":99999}");

    // when:
        ResponseOptions response = given().spec(request)
                .put("/fraudcheck");

    // then:
        assertThat(response.statusCode()).isEqualTo(200);
        assertThat(response.header("Content-
Type")).matches("application/vnd.fraud.v1.json.*");
    // and:
        DocumentContext parsedJson =
JsonPath.parse(response.getBody().asString());

assertThatJson(parsedJson).field("fraudCheckStatus").matc
hes("[A-Z]{5}");

assertThatJson(parsedJson).field("rejectionReason").isEqu
alTo("Amount too high");
```

```
}
```

您可以看到 `value(consumer(…), producer(…))` 块中存在的所有 `producer()` 部分合同注入测试。

重要的是在生产者方面，我们也在做 TDD。我们有一个测试形式的期望。此测试正在向我们自己的应用程序拍摄一个在合同中定义的 URL，标题和主体的请求。它也期待响应中非常精确地定义的值。换句话说，您是 `red` `green` 和 `refactor` 的 `red` 部分。将 `red` 转换为 `green` 的时间。

写入缺少的实现

现在，由于我们现在预期的输入和预期的输出是什么，我们来写这个缺少的实现。

```
@RequestMapping(
            value = "/fraudcheck",
            method = PUT,
            consumes = FRAUD_SERVICE_JSON_VERSION_1,
            produces = FRAUD_SERVICE_JSON_VERSION_1)
public FraudCheckResult fraudCheck(@RequestBody
FraudCheck fraudCheck) {
if (amountGreaterThanThreshold(fraudCheck)) {
      return new FraudCheckResult(FraudCheckStatus.FRAUD,
AMOUNT_TOO_HIGH);
}
return new FraudCheckResult(FraudCheckStatus.OK,
NO_REASON);
}
```

如果再次执行 `./mvnw clean install`，测试将通过。由于 `Spring Cloud Contract Verifier` 插件将测试添加到 `generated-test-sources`，您可以从 IDE 中实际运行这些测试。

部署你的应用程序

完成工作后，现在是部署变更的时候了。首先合并分支

```
git checkout master
git merge --no-ff contract-change-pr
git push origin master
```

那么我们假设你的 CI 将像`./mvnw clean deploy`一样运行，它将发布应用程序和存根工件。

消费方（贷款发行）最后一步

作为贷款发行服务（欺诈检测服务器的消费者）的开发人员：

合并分支到主

```
git checkout master
git merge --no-ff contract-change-pr
```

在线工作

现在，您可以禁用 Spring Cloud Contract Stub Runner 广告的离线工作，以提供存储库与存根的位置。此时，服务器端的存根将自动从 Nexus / Artifactory 下载。您可以关闭注释中的`workOffline`参数的值。在下面你可以看到一个通过改变属性实现相同的例子。

```
stubrunner:
  ids: 'com.example:http-server-dsl:+:stubs:8080'
  repositoryRoot: http://repo.spring.io/libs-snapshot
```

就是这样！

依赖

添加依赖关系的最佳方法是使用正确的 `starter` 依赖关系。

对于 `stub-runner` 使用 `spring-cloud-starter-stub-runner`，当您使用插件时，只需添加 `spring-cloud-starter-contract-verifier`。

附加链接

以下可以找到与 Spring Cloud Contract 验证器和 Stub Runner 相关的一些资源。注意，有些可以过时，因为 Spring Cloud Contract 验证程序项目正在不断发展。

阅读

- 来自 Marcin Grzejszczak 关于 Accurest 的演讲
- 来自 Marcin Grzejszczak 的博客的 Accurest 相关文章
- 来自 Marcin Grzejszczak 博客的 Spring Cloud Contract 相关文章
- Groovy 关于 JSON 的文档

样品

在这里可以找到一些样品。

常问问题

为什么使用 Spring Cloud Contract 验证器而不是 X?

目前 Spring Cloud Contract 验证器是基于 JVM 的工具。因此，当您已经为 JVM 创建软件时，可能是您的第一选择。这个项目有很多非常有趣的功能，但特别是其中一些绝对让 Spring Cloud Contract Verifier 在消费者驱动合同（CDC）工具的“市场”上脱颖而出。许多最有趣的是：

- CDC 可以通过消息传递
- 清晰易用，静态 DSL
- 可以将当前的 JSON 文件粘贴到合同中，并且仅编辑其元素
- 从定义的合同自动生成测试
- Stub Runner 功能 - 存根在运行时自动从 Nexus / Artifactory 下载
- Spring Cloud 集成 - 集成测试不需要发现服务

这个值是（consumer（），producer（））？

与存根相关的最大挑战之一是可重用性。只有如果他们能够被广泛使用，他们是否会服务于他们的目的。通常使得难点是请求/响应元素的硬编码值。例如日期或 ids。想象下面的 JSON 请求

```
{
    "time" : "2016-10-10 20:10:15",
    "id" : "9febab1c-6f36-4a0b-88d6-3b6a6d81cd4a",
    "body" : "foo"
}
```

和 JSON 响应

```
{
    "time" : "2016-10-10 21:10:15",
```

```
    "id" : "c4231e1f-3ca9-48d3-b7e7-567d55f0d051",
    "body" : "bar"
}
```

想象一下，通过更改系统中的时钟或提供数据提供者的存根实现，设置`time`字段的正确值（我们假设这个内容是由数据库生成的）所需的痛苦。这同样涉及到称为`id`的字段。你会创建一个 UUID 发生器的 stubbed 实现？没有意义

所以作为一个消费者，你想发送一个匹配任何形式的时间或任何 UUID 的请求。这样，您的系统将照常工作 - 将生成数据，您不必将任何东西存入。假设在上述 JSON 的情况下，最重要的部分是`body`字段。您可以专注于其他领域，并提供匹配。换句话说，你想要的存根是这样工作的：

```
{
    "time" : "SOMETHING THAT MATCHES TIME",
    "id" : "SOMETHING THAT MATCHES UUID",
    "body" : "foo"
}
```

就响应作为消费者而言，您需要具有可操作性的具体价值。所以这样的 JSON 是有效的

```
{
    "time" : "2016-10-10 21:10:15",
    "id" : "c4231e1f-3ca9-48d3-b7e7-567d55f0d051",
    "body" : "bar"
}
```

从前面的部分可以看出，我们从合同中产生测试。所以从生产者的角度看，情况看起来差别很大。我们正在解析提供的合同，在测试中我们想向您的端点发送一个真正的请求。因此，对于请求的生产者来说，我们不能进行任何匹配。我们需要具体的价值观，使制片人的后台能够工作。这样的 JSON 将是一个有效的：

```
{
    "time" : "2016-10-10 20:10:15",
    "id" : "9febab1c-6f36-4a0b-88d6-3b6a6d81cd4a",
    "body" : "foo"
}
```

另一方面，从合同的有效性的角度来看，响应不一定必须包含 `time` 或 `id` 的具体值。假设您在生产者方面产生这些 - 再次，您必须做很多桩，以确保始终返回相同的值。这就是为什么从生产者那边你可能想要的是以下回应：

```
{
    "time" : "SOMETHING THAT MATCHES TIME",
    "id" : "SOMETHING THAT MATCHES UUID",
    "body" : "bar"
}
```

那么您如何才能为消费者提供一次匹配，并为生产者提供具体的价值，反之亦然？在 Spring Cloud Contract 中，我们允许您提供**动态值**。这意味着通信双方可能有所不同。你可以传递值：

可以通过 `value` 方法

```
value(consumer(...), producer(...))
value(stub(...), test(...))
value(client(...), server(...))
```

或使用`$()`方法

```
$(consumer(...), producer(...))
$(stub(...), test(...))
$(client(...), server(...))
```

您可以在 Contract DSL 部分阅读更多信息。

调用`value()`或`$()`告诉 Spring Cloud Contract 您将传递一个动态值。在`consumer()`方法中，传递消费者端（在生成的存根）中应该使用的值。在`producer()`方法中，传递应在生产者端使用的值（在生成的测试中）。

提示	如果一方面你已经通过了正则表达式，而你没有通过另一方，那么对方就会自动生

大多数情况下，您将使用该方法与`regex`辅助方法。例如`consumer(regex('[0-9]{10}'))`。

总而言之，上述情景的合同看起来或多或少是这样的（正则表达式的时间和 UUID 被简化，很可能是无效的，但是我们希望在这个例子中保持很简单）：

```
org.springframework.cloud.contract.spec.Contract.make {
                    request {
                          method 'GET'
                          url '/someUrl'
                          body([
                              time :
value(consumer(regex('[0-9]{4}-[0-9]{2}-[0-9]{2} [0-2][0-9]-[0-5][0-9]-[0-5][0-9]')),
                              id:
value(consumer(regex('[0-9a-zA-z]{8}-[0-9a-zA-z]{4}-[0-9a-zA-z]{4}-[0-9a-zA-z]{12}'))
                              body: "foo"
                          ])
                    }
              response {
                    status 200
                    body([
                              time :
value(producer(regex('[0-9]{4}-[0-9]{2}-[0-9]{2} [0-2][0-9]-[0-5][0-9]-[0-5][0-9]')),
                              id:
value([producer(regex('[0-9a-zA-z]{8}-[0-9a-zA-z]{4}-[0-9a-zA-z]{4}-[0-9a-zA-z]{12}'))
                              body: "bar"
                          ])
```

```
                        }
}
```

重要	请阅读与 JSON 相关的 Groovy 文档，以了解如何正确构建请求/响应实体。

如何做 Stubs 版本控制?

API 版本控制

让我们尝试回答一个真正意义上的版本控制的问题。如果你指的是 API 版本，那么有不同的方法。

- 使用超媒体，链接，不要通过任何方式版本您的 API
- 通过标题/网址传递版本

我不会试图回答一个方法更好的问题。无论适合您的需求，并允许您创造商业价值应被挑选。

假设你做你的 API 版本。在这种情况下，您应该提供与您支持的许多版本一样多的合同。您可以为每个版本创建一个子文件夹，或将其附加到合同名称 - 无论如何适合您。

JAR 版本控制

如果通过版本控制是指包含存根的 JAR 的版本，那么基本上有两种主要方法。

假设您正在进行连续交付/部署，这意味着您每次通过管道生成新版本的 jar 时，该 jar 可以随时进行生产。例如你的 jar 版本看起来像这样（它建立在 20.10.2016 在 20:15:21）：

```
1.0.0.20161020-201521-RELEASE
```

在这种情况下，您生成的存根 jar 将看起来像这样。

```
1.0.0.20161020-201521-RELEASE-stubs.jar
```

在这种情况下，您应该在 `application.yml` 或 `@AutoConfigureStubRunner` 内引用存根提供最新版本的存根。您可以通过传递`+`号来做到这一点。例

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:8080"})
```

如果版本控制是固定的（例如 `1.0.4.RELEASE` 或 `2.1.1`），则必须设置 jar 版本的具体值。示例 2.1.1。

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:2.1.1:stubs:8080"})
```

开发者或生产者存根

您可以操作分类器，以针对其他服务的存根或部署到生产的存根的当前开发版本来运行测试。如果您在构建生产部署之后，使用 `prod-stubs` 分类器来更改构建部署，那么您可以在一个案例中使用 dev stub 运行测试，另一个则使用 prod stub 进行测试。

使用开发版存根的测试示例

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:stubs:8080"})
```

使用生产版本的存根的测试示例

```
@AutoConfigureStubRunner(ids = {"com.example:http-server-dsl:+:prod-stubs:8080"})
```

您也可以通过部署管道中的属性传递这些值。

共同回购合同

存储合同以外的另一种方法是将它们保存在一个共同的地方。它可能与消费者无法克隆生产者代码的安全问题相关。另外，如果您在一个地方保留合约，那么作为生产者，您将知道有多少消费者，以及您的本地变更会消费哪些消费者。

回购结构

假设我们有一个坐标为 `com.example:server` 和 3 个消费者的生产者：`client1`，`client2`，`client3`。然后在具有常见合同的存储库中，您将具有以下设置（您可以在此处查看：

```
├── com
│   └── example
│       └── server
│           ├── client1
│           │   └── expectation.groovy
│           ├── client2
│           │   └── expectation.groovy
│           ├── client3
│           │   └── expectation.groovy
│           └── pom.xml
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
    └── assembly
        └── contracts.xml
```

您可以看到下面的斜线分隔的 groupid `/`工件 id 文件夹（`com/example/server`），您对 3 个消费者（`client1`，`client2` 和 `client3`）有期望。期望是本文档中描述的标准 Groovy DSL 合同文件。该存储库必须生成一个将一对一映射到回收内容的 JAR 文件。

`server` 文件夹内的 `pom.xml` 示例。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>

	<groupId>com.example</groupId>
	<artifactId>server</artifactId>
	<version>0.0.1-SNAPSHOT</version>

	<name>Server Stubs</name>
	<description>POM used to install locally stubs for consumer side</description>

	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>1.5.0.BUILD-SNAPSHOT</version>
		<relativePath />
	</parent>

	<properties>
		<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
		<java.version>1.8</java.version>
		<spring-cloud-contract.version>1.1.0.BUILD-SNAPSHOT</spring-cloud-contract.version>
		<spring-cloud-dependencies.version>Dalston.BUILD-SNAPSHOT</spring-cloud-dependencies.version>
```

```
        <excludeBuildFolders>true</excludeBuildFolders>
        </properties>

        <dependencyManagement>
                <dependencies>
                        <dependency>

        <groupId>org.springframework.cloud</groupId>
                                <artifactId>spring-cloud-
dependencies</artifactId>
                                <version>${spring-cloud-
dependencies.version}</version>
                                <type>pom</type>
                                <scope>import</scope>
                        </dependency>
                </dependencies>
        </dependencyManagement>

        <build>
                <plugins>
                        <plugin>

        <groupId>org.springframework.cloud</groupId>
                                <artifactId>spring-cloud-
contract-maven-plugin</artifactId>
                                <version>${spring-cloud-
contract.version}</version>
                                <extensions>true</extensions>
                                <configuration>
                                        <!-- By default it would
search under src/test/resources/ -->

        <contractsDirectory>${project.basedir}</contractsDi
rectory>
                                </configuration>
                        </plugin>
                </plugins>
        </build>

        <repositories>
                <repository>
                        <id>spring-snapshots</id>
                        <name>Spring Snapshots</name>
```

```
<url>https://repo.spring.io/snapshot</url>
            <snapshots>
                  <enabled>true</enabled>
            </snapshots>
      </repository>
      <repository>
            <id>spring-milestones</id>
            <name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>
            <snapshots>
                  <enabled>false</enabled>
            </snapshots>
      </repository>
      <repository>
            <id>spring-releases</id>
            <name>Spring Releases</name>

<url>https://repo.spring.io/release</url>
            <snapshots>
                  <enabled>false</enabled>
            </snapshots>
      </repository>
</repositories>
<pluginRepositories>
      <pluginRepository>
            <id>spring-snapshots</id>
            <name>Spring Snapshots</name>

<url>https://repo.spring.io/snapshot</url>
            <snapshots>
                  <enabled>true</enabled>
            </snapshots>
      </pluginRepository>
      <pluginRepository>
            <id>spring-milestones</id>
            <name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>
            <snapshots>
                  <enabled>false</enabled>
            </snapshots>
      </pluginRepository>
```

```
            <pluginRepository>
                  <id>spring-releases</id>
                  <name>Spring Releases</name>

      <url>https://repo.spring.io/release</url>
                  <snapshots>
                        <enabled>false</enabled>
                  </snapshots>
            </pluginRepository>
      </pluginRepositories>

</project>
```

你可以看到除了 Spring Cloud Contract Maven 插件之外没有依赖关系。这些垃圾是消费者运行 `mvn clean install -DskipTests` 来本地安装生产者项目的存根的必要条件。

根文件夹中的 `pom.xml` 可以如下所示:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>

      <groupId>com.example.standalone</groupId>
      <artifactId>contracts</artifactId>
      <version>0.0.1-SNAPSHOT</version>

      <name>Contracts</name>
      <description>Contains all the Spring Cloud
Contracts, well, contracts. JAR used by the producers to
generate tests and stubs</description>

      <properties>
            <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
      </properties>
```

```
	<build>
		<plugins>
			<plugin>

	<groupId>org.apache.maven.plugins</groupId>
				<artifactId>maven-assembly-
plugin</artifactId>
				<executions>
					<execution>
						<id>contracts</id>
						<phase>prepare-
package</phase>
						<goals>

	<goal>single</goal>
						</goals>
						<configuration>

	<attach>true</attach>

	<descriptor>${basedir}/src/assembly/contracts.xml</
descriptor>
							<!-- If you
want an explicit classifier remove the following line -->

	<appendAssemblyId>false</appendAssemblyId>
						</configuration>
					</execution>
				</executions>
			</plugin>
		</plugins>
	</build>

</project>
```

它正在使用程序集插件来构建所有合同的 JAR。此类设置的示例如下:

```
<assembly xmlns="http://maven.apache.org/plugins/maven-
assembly-plugin/assembly/1.1.3"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/plugins/maven
```

```
-assembly-plugin/assembly/1.1.3
http://maven.apache.org/xsd/assembly-1.1.3.xsd">
	<id>project</id>
	<formats>
		<format>jar</format>
	</formats>
	<includeBaseDirectory>false</includeBaseDirectory>
	<fileSets>
		<fileSet>

	<directory>${project.basedir}</directory>
			<outputDirectory>/</outputDirectory>

	<useDefaultExcludes>true</useDefaultExcludes>
			<excludes>

	<exclude>**/${project.build.directory}/**</exclude>
				<exclude>mvnw</exclude>
				<exclude>mvnw.cmd</exclude>
				<exclude>.mvn/**</exclude>
				<exclude>src/**</exclude>
			</excludes>
		</fileSet>
	</fileSets>
</assembly>
```

工作流程

工作流程将与 `Step by step guide to CDC` 中提供的工作流程类似。唯一的区别是生产者不再拥有合同。所以消费者和生产者必须在共同的仓库中处理共同的合同。

消费者

当**消费者**希望脱机工作，而不是克隆生产者代码时，消费者团队克隆了公用存储库，转到所需的生产者的文件夹（例如 `com/example/server`），并运行 `mvn clean install -DskipTests` 在本地安装存根从合同转换。

提示	您需要在本地安装 Maven

制片人

作为一个**生产者**，足以改变 Spring Cloud Contract 验证器来提供包含合同的 URL 和依赖关系：

```
<plugin>
	<groupId>org.springframework.cloud</groupId>
	<artifactId>spring-cloud-contract-maven-
plugin</artifactId>
	<configuration>

	<contractsRepositoryUrl>http://link/to/your/nexus/o
r/artifactory/or/sth</contractsRepositoryUrl>
		<contractDependency>

	<groupId>com.example.standalone</groupId>
			<artifactId>contracts</artifactId>
		</contractDependency>
	</configuration>
</plugin>
```

使用此设置，将从 http://link/to/your/nexus/or/artifactory/or/sth 下载具有 groupid `com.example.standalone` 和 artifactid `contracts` 的 JAR。然后将在本地临时文件夹中解压缩，并将 `com/example/server` 下的合同作为用于生成测试和存根的选择。由于这个惯例，生产者团队将会知道当一些不兼容的更改完成时，哪些消费者团队将被破坏。

其余的流程看起来是一样的。

我可以有多个基类进行测试吗？

是! 查看 Gradle 或 Maven 插件的合同部分的不同基类。

如何调试生成的测试客户端发送的请求/响应?

生成的测试都以某种形式或时尚的方式依赖于 Apache HttpClient 进行 RestAssured。HttpClient 有一个名为 wire logging 的工具，它将整个请求和响应记录到 HttpClient。Spring Boot 有一个日志记录通用应用程序属性来做这种事情，只需将其添加到应用程序属性中即可

```
logging.level.org.apache.http.wire=DEBUG
```

可以从响应中引用请求吗?

是! 使用版本 1.1.0，我们添加了这样一种可能性。在 HTTP 存根服务器端，我们正在为 WireMock 提供支持。在其他 HTTP 服务器存根的情况下，您必须自己实现该方法。

Spring Cloud Contract 验证者 HTTP 毕业项目

先决条件

为了在 WireMock 中使用 Spring Cloud Contract 验证器，您必须使用 Gradle 或 Maven 插件。

警告	如果您想在项目中使用 Spock，则必须单独添加 `spock-core` 和 `spock-spring`

添加具有依赖关系的渐变插件

```
buildscript {
	repositories {
		mavenCentral()
	}
	dependencies {
	    classpath "org.springframework.boot:spring-boot-gradle-plugin:${springboot_version}"
		classpath "org.springframework.cloud:spring-cloud-contract-gradle-plugin:${verifier_version}"
	}
}

apply plugin: 'groovy'
apply plugin: 'spring-cloud-contract'

dependencyManagement {
	imports {
		mavenBom "org.springframework.cloud:spring-cloud-contract-dependencies:${verifier_version}"
	}
}

dependencies {
	testCompile 'org.codehaus.groovy:groovy-all:2.4.6'
	// example with adding Spock core and Spock Spring
	testCompile 'org.spockframework:spock-core:1.0-groovy-2.4'
	testCompile 'org.spockframework:spock-spring:1.0-groovy-2.4'
	testCompile 'org.springframework.cloud:spring-cloud-starter-contract-verifier'
}
```

Gradle 的快照版本

将其他快照存储库添加到您的 build.gradle 以使用快照版本，每次成功构建后都会自动上传:

```
buildscript {
```

```
	repositories {
		mavenCentral()
		mavenLocal()
		maven { url
"http://repo.spring.io/snapshot" }
		maven { url
"http://repo.spring.io/milestone" }
		maven { url
"http://repo.spring.io/release" }
	}
}
```

添加存根

默认情况下 Spring Cloud Contract 验证器正在

`src/test/resources/contracts` 目录中查找存根。

包含存根定义的目录被视为一个类名称，每个存根定义被视为单个测试。我们假设它至少包含一个用作测试类名称的目录。如果有多个级别的嵌套目录，除了最后一个级别将被用作包名称。所以具有以下结构

```
src/test/resources/contracts/myservice/shouldCreateUser.groovy
src/test/resources/contracts/myservice/shouldReturnUser.groovy
```

Spring Cloud Contract 验证程序将使用两种方法创建测试类
`defaultBasePackage.MyService`

- `shouldCreateUser()`
- `shouldReturnUser()`

运行插件

插件注册自己在 `check` 任务之前被调用。只要您希望它成为构建过程的一部分，您就无所事事。如果您只想生成测试，请调用 `generateContractTests` 任务。

默认设置

默认的 Gradle 插件设置创建了以下 Gradle 部分的构建（它是一个伪代码）

```
contracts {
    targetFramework = 'JUNIT'
    testMode = 'MockMvc'
    generatedTestSourcesDir =
project.file("${project.buildDir}/generated-test-
sources/contracts")
    contractsDslDir =
"${project.rootDir}/src/test/resources/contracts"
    basePackageForTests =
'org.springframework.cloud.verifier.tests'
    stubsOutputDir =
project.file("${project.buildDir}/stubs")

    // the following properties are used when you want to
provide where the JAR with contract lays
    contractDependency {
        stringNotation = ''
    }
    contractsPath = ''
    contractsWorkOffline = false
}

tasks.create(type: Jar, name: 'verifierStubsJar',
dependsOn: 'generateClientStubs') {
    baseName = project.name
    classifier = contracts.stubsSuffix
    from contractVerifier.stubsOutputDir
}

project.artifacts {
    archives task
}
```

```
tasks.create(type: Copy, name: 'copyContracts') {
    from contracts.contractsDslDir
    into contracts.stubsOutputDir
}

verifierStubsJar.dependsOn 'copyContracts'

publishing {
    publications {
        stubs(MavenPublication) {
            artifactId project.name
            artifact verifierStubsJar
        }
    }
}
```

配置插件

要更改默认配置，只需在您的 Gradle 配置中添加 `contracts` 代码段即可

```
contracts {
      testMode = 'MockMvc'
      baseClassForTests = 'org.mycompany.tests'
      generatedTestSourcesDir =
project.file('src/generatedContract')
}
```

配置选项

- **testMode** - 定义接受测试的模式。默认的基于 Spring 的 MockMvc 的 MockMvc。也可以将其更改为 **JaxRsClient** 或**显式**为真实的 HTTP 调用。
- **导入** - 应包含在生成的测试中的**导入的**数组（例如 ['org.myorg.Matchers']）。默认为空数组[]
- **staticImports** - 应该包含在生成的测试中的静态导入的数组（例如 ['org.myorg.Matchers。*']）。默认为空数组[]

- **basePackageForTests** - 为所有生成的测试指定基础包。默认设置为 org.springframework.cloud.verifier.tests
- **baseClassForTests** - 所有生成的测试的基类。如果使用 Spock 测试，默认为 `spock.lang.Specification`。
- **packageWithBaseClasses** - 而不是为基类提供固定值，您可以提供一个所有基类放置的包。优先于 **baseClassForTests**。
- **baseClassMappings** - 明确地将合约包映射到基类的 FQN。优先于 **packageWithBaseClasses** 和 **baseClassForTests**。
- **ruleClassForTests** - 指定应该添加到生成的测试类的规则。
- **ignoredFiles** - Ant 匹配器，允许定义要跳过哪些处理的存根文件。默认为空数组[]
- **contractsDslDir** - 包含使用 GroovyDSL 编写的合同的目录。默认 `$rootDir/src/test/resources/contracts`
- **generatedTestSourcesDir** - 应该放置从 Groovy DSL **生成**测试的测试源目录。默认`$buildDir/generated-test-sources/contractVerifier`
- **stubsOutputDir** - 应该放置从 Groovy DSL 生成的 WireMock 存根的目录
- **targetFramework** - 要使用的目标测试框架; JUnit 作为默认框架，目前支持 Spock 和 JUnit

当您希望提供合同所在 JAR 的位置时，将使用以下属性

- **contractDependency** - 提供 `groupid:artifactid:version:classifier` 坐标的依赖关系。您可以使用 `contractDependency` 关闭来设置它
- **contractPath** - 如果下载合同部分将默认为 `groupid/artifactid`，其中 `groupid` 将被分隔。否则将扫描提供的目录下的合同
- **contractsWorkOffline** - 为了不下载依赖关系，每次下载一次，然后离线工作（重用本地 Maven repo）

所有测试的单一基类

在默认的 MockMvc 中使用 Spring Cloud Contract 验证器时，您需要为所有生成的验收测试创建一个基本规范。在这个类中，您需要指向应验证的端点。

```
abstract class BaseMockMvcSpec extends Specification {

	def setup() {
		RestAssuredMockMvc.standaloneSetup(new
PairIdController())
	}

	void isProperCorrelationId(Integer correlationId) {
		assert correlationId == 123456
	}

	void isEmpty(String value) {
		assert value == null
	}

}
```

在使用 `Explicit` 模式的情况下，您可以像普通集成测试一样使用基类来初始化整个测试的应用程序。在 `JAXRSCLIENT` 模式的情况下，这个基类也应该包含

`protected WebTarget webTarget` 字段，现在测试 JAX-RS API 的唯一选项是启动 Web 服务器。

不同的基础类别的合同

如果您的基类在合同之间不同，您可以告诉 Spring Cloud Contract 插件哪个类应该由自动生成测试扩展。你有两个选择：

- 遵循约定，提供 `packageWithBaseClasses`
- 通过 `baseClassMappings` 提供显式映射

惯例

约定是这样的，如果你有合同，例如 `src/test/resources/contract/foo/bar/baz/`，并将 `packageWithBaseClasses` 属性的值提供给 `com.example.base`，那么我们将假设 `com.example.base` 下有一个 `BarBazBase` 类包。换句话说，如果它们存在并且形成具有 `Base` 后缀的类，那么我们将使用最后两个包的部分。优先于 **baseClassForTests**。`contracts` 关闭中的使用示例：

```
packageWithBaseClasses = 'com.example.base'
```

制图

您可以手动将合同包的正则表达式映射为匹配合同的基类的完全限定名称。我们来看看下面的例子：

```
baseClassForTests = "com.example.FooBase"
```

```
baseClassMappings {
        baseClassMapping('.*/com/.*',
'com.example.ComBase')
        baseClassMapping('.*/bar/.*':'com.example.BarBase')
}
```

我们假设你有合同

- `src/test/resources/contract/com/` - `src/test/resources/contract/foo/`

通过提供 `baseClassForTests`，我们有一个后备案例，如果映射没有成功（您也可以提供 `packageWithBaseClasses` 作为备用）。这样，从 `src/test/resources/contract/com/`合同产生的测试将扩展 `com.example.ComBase`，而其余的测试将扩展 `com.example.FooBase`。

调用生成的测试

为确保提供方对定义的合同进行投诉，您需要调用：

```
./gradlew generateContractTests test
```

Spring Cloud Contract 消费者验证者

在消费者服务中，您需要以与提供商相同的方式配置 Spring Cloud Contract 验证器插件。如果您不想使用 Stub Runner，则需要复制存储在 `src/test/resources/contracts` 中的合同，并使用以下命令生成 WireMock json 存根：

```
./gradlew generateClientStubs
```

请注意，必须为存根生成设置 `stubsOutputDir` 选项才能正常工作。

当存在时，json 存根可用于消费者自动测试。

```
@ContextConfiguration(loader ==
SpringApplicationContextLoader, classes == Application)
class LoanApplicationServiceSpec extends Specification {

 @ClassRule
 @Shared
 WireMockClassRule wireMockRule == new
WireMockClassRule()

 @Autowired
 LoanApplicationService sut

 def 'should successfully apply for loan'() {
   given:
       LoanApplication application =
                    new LoanApplication(client: new
Client(clientPesel: '12345678901'), amount: 123.123)
   when:
       LoanApplicationResult loanApplication ==
sut.loanApplication(application)
   then:
       loanApplication.loanApplicationStatus ==
LoanApplicationStatus.LOAN_APPLIED
       loanApplication.rejectionReason == null
 }
}
```

在 LoanApplication 下面调用 FraudDetection 服务。此请求由使用由 Spring Cloud Contract 验证器生成的存根配置的 WireMock 服务器处理。

在您的 Maven 项目中使用

添加 maven 插件

添加 Spring Cloud Contract BOM

```
<dependencyManagement>
```

```
        <dependencies>
                <dependency>

        <groupId>org.springframework.cloud</groupId>
                        <artifactId>spring-cloud-
dependencies</artifactId>
                        <version>${spring-cloud-
dependencies.version}</version>
                        <type>pom</type>
                        <scope>import</scope>
                </dependency>
        </dependencies>
</dependencyManagement>
```

接下来，`Spring Cloud Contract Verifier` Maven 插件

```
<plugin>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-contract-maven-
plugin</artifactId>
        <version>${spring-cloud-contract.version}</version>
        <extensions>true</extensions>
        <configuration>

        <packageWithBaseClasses>com.example.fraud</packageW
ithBaseClasses>
        </configuration>
</plugin>
```

您可以在 Spring Cloud Contract Maven 插件文档中阅读更多内容

Maven 的快照版本

对于快照/里程碑版本，您必须将以下部分添加到您的 `pom.xml`

```
<repositories>
        <repository>
                <id>spring-snapshots</id>
                <name>Spring Snapshots</name>
                <url>https://repo.spring.io/snapshot</url>
                <snapshots>
```

```
                        <enabled>true</enabled>
                </snapshots>
        </repository>
        <repository>
                <id>spring-milestones</id>
                <name>Spring Milestones</name>
                <url>https://repo.spring.io/milestone</url>
                <snapshots>
                        <enabled>false</enabled>
                </snapshots>
        </repository>
        <repository>
                <id>spring-releases</id>
                <name>Spring Releases</name>
                <url>https://repo.spring.io/release</url>
                <snapshots>
                        <enabled>false</enabled>
                </snapshots>
        </repository>
</repositories>
<pluginRepositories>
        <pluginRepository>
                <id>spring-snapshots</id>
                <name>Spring Snapshots</name>
                <url>https://repo.spring.io/snapshot</url>
                <snapshots>
                        <enabled>true</enabled>
                </snapshots>
        </pluginRepository>
        <pluginRepository>
                <id>spring-milestones</id>
                <name>Spring Milestones</name>
                <url>https://repo.spring.io/milestone</url>
                <snapshots>
                        <enabled>false</enabled>
                </snapshots>
        </pluginRepository>
        <pluginRepository>
                <id>spring-releases</id>
                <name>Spring Releases</name>
                <url>https://repo.spring.io/release</url>
                <snapshots>
                        <enabled>false</enabled>
                </snapshots>
```

```
        </pluginRepository>
</pluginRepositories>
```

添加存根

默认情况下 Spring Cloud Contract 验证器正在 `src/test/resources/contracts` 目录中查找存根。包含存根定义的目录被视为一个类名称，每个存根定义被视为单个测试。我们假设它至少包含一个用作测试类名称的目录。如果有多个级别的嵌套目录，除了最后一个级别将被用作包名称。所以具有以下结构

```
src/test/resources/contracts/myservice/shouldCreateUser.groovy
src/test/resources/contracts/myservice/shouldReturnUser.groovy
```

Spring Cloud Contract 验证者将使用两种方法创建测试类 `defaultBasePackage.MyService` - `shouldCreateUser()` - `shouldReturnUser()`

运行插件

插件目标 `generateTests` 被分配为阶段 `generate-test-sources`。只要您希望它成为构建过程的一部分，您就无所事事。如果您只想生成测试，请调用 `generateTests` 目标。

配置插件

要更改默认配置，只需将 `configuration` 部分添加到插件定义或 `execution` 定义。

```
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-
plugin</artifactId>
    <executions>
        <execution>
            <goals>
                <goal>convert</goal>
                <goal>generateStubs</goal>
                <goal>generateTests</goal>
            </goals>
        </execution>
    </executions>
    <configuration>

<basePackageForTests>org.springframework.cloud.verifier.t
witter.place</basePackageForTests>

<baseClassForTests>org.springframework.cloud.verifier.twi
tter.place.BaseMockMvcSpec</baseClassForTests>
    </configuration>
</plugin>
```

重要配置选项

- **testMode** - 定义接受测试的模式。默认 `MockMvc`，它基于 Spring 的 MockMvc。对于真正的 HTTP 呼叫，它也可以更改为 `JaxRsClient` 或 `Explicit`。
- **basePackageForTests** - 为所有生成的测试指定基础包。默认设置为 `org.springframework.cloud.verifier.tests`。
- **ruleClassForTests** - 指定应该添加到生成的测试类的规则。
- **baseClassForTests** - 生成测试的基类。如果使用 Spock 测试，默认为 `spock.lang.Specification`。

- **contractDir** - 包含使用 GroovyDSL 编写的合同的目录。默认 `/src/test/resources/contracts`。
- **testFramework** - 要使用的目标测试框架; JUnit 作为默认框架，目前支持 Spock 和 JUnit
- **packageWithBaseClasses** - 而不是为基类提供固定值，您可以提供一个所有基类放置的包。约定是这样的，如果你有合同 `src/test/resources/contract/foo/bar/baz/`，并提供这个属性的值到 `com.example.base`，那么我们将假设 `com.example.base` 包含 `com.example.base` 类。优先于 **baseClassForTests**
- **baseClassMappings** - 您必须提供 `contractPackageRegex` 的基类映射列表，该列表根据合同所在的包进行检查，并且 `baseClassFQN` 映射到匹配合同的基类的完全限定名称。如果您有合同 `src/test/resources/contract/foo/bar/baz/`并映射了属性`.*`→`com.example.base.BaseClass`，则从这些合同生成的测试类将扩展 `com.example.base.BaseClass`。优先于 **packageWithBaseClasses** 和 **baseClassForTests**。

如果要从 Maven 存储库中下载合同定义，可以使用

- **contractsRepositoryUrl** - 具有合同的工件的 repo 的 URL（如果没有提供）应使用当前的 Maven
- **contractDependency** - 包含所有打包合同的合同依赖关系

- **contractPath** - 通过打包合同在 JAR 中具体合同的路径。默认为 `groupid/artifactid`，其中 `gropuid` 被斜杠分隔。
- **contractWorkOffline** - 如果依赖关系应该被下载，或者本地 Maven 只能被重用

有关完整信息，请参阅插件文档

所有测试的单一基类

在默认的 MockMvc 中使用 Spring Cloud Contract 验证器时，您需要为所有生成的验收测试创建一个基本规范。在这个类中，您需要指向应验证的端点。

```
package org.mycompany.tests

import org.mycompany.ExampleSpringController
import
com.jayway.restassured.module.mockmvc.RestAssuredMockMvc
import spock.lang.Specification

class  MvcSpec extends Specification {
  def setup() {
   RestAssuredMockMvc.standaloneSetup(new
ExampleSpringController())
  }
}
```

在使用 `Explicit` 模式的情况下，您可以像常规集成测试一样使用基类来初始化整个测试的应用程序。在 `JAXRSCLIENT` 模式的情况下，这个基类也应该包含 `protected WebTarget webTarget` 字段，现在测试 JAX-RS API 的唯一选项是启动 Web 服务器。

不同的基础类别的合同

如果您的基类在合同之间不同，您可以告诉 Spring Cloud Contract 插件哪个类应该由自动生成测试扩展。你有两个选择：

- 遵循约定，提供 `packageWithBaseClasses`
- 通过 `baseClassMappings` 提供显式映射

惯例

约定是这样的，如果你有合同，例如 `src/test/resources/contract/hello/v1/`，并将 `packageWithBaseClasses` 属性的值提供给 `hello`，那么我们将假设在 `hello` 下有一个 `HelloV1Base` 类包。换句话说，如果它们存在并且形成具有 `Base` 后缀的类，那么我们将使用最后两个包的部分。优先于 **baseClassForTests**。使用示例：

```
<plugin>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-maven-
plugin</artifactId>
      <configuration>

      <packageWithBaseClasses>hello</packageWithBaseClass
es>
      </configuration>
</plugin>
```

制图

您可以手动将合同包的正则表达式映射为匹配合同的基类的完全限定名称。您必须提供 `baseClassMappingsbaseClassMapping` 的

`contractPackageRegex` 列表 `contractPackageRegex` 到 `baseClassFQN` 映射。我们来看看下面的例子：

```
<plugin>
	<groupId>org.springframework.cloud</groupId>
	<artifactId>spring-cloud-contract-maven-plugin</artifactId>
	<configuration>

	<baseClassForTests>com.example.FooBase</baseClassForTests>
		<baseClassMappings>
			<baseClassMapping>

	<contractPackageRegex>.*com.*</contractPackageRegex>


	<baseClassFQN>com.example.TestBase</baseClassFQN>
			</baseClassMapping>
		</baseClassMappings>
	</configuration>
</plugin>
```

我们假设你有合同

- `src/test/resources/contract/com/` - `src/test/resources/contract/foo/`

通过提供 `baseClassForTests`，我们有一个后备程序，如果映射没有成功（你也可以提供 `packageWithBaseClasses` 作为备用）。这样，从 `src/test/resources/contract/com/` 合同生成的测试将扩展 `com.example.ComBase`，而其余的测试将扩展 `com.example.FooBase`。

调用生成的测试

Spring Cloud Contract Maven 插件将验证码生成到目录`/generated-test-sources/contractVerifier`中，并将此目录附加到`testCompile`目标。

对于 Groovy Spock 代码使用:

```
<plugin>
	<groupId>org.codehaus.gmavenplus</groupId>
	<artifactId>gmavenplus-plugin</artifactId>
	<version>1.5</version>
	<executions>
		<execution>
			<goals>
				<goal>testCompile</goal>
			</goals>
		</execution>
	</executions>
	<configuration>
		<testSources>
			<testSource>

	<directory>${project.basedir}/src/test/groovy</dire
ctory>
				<includes>

	<include>**/*.groovy</include>
				</includes>
			</testSource>
			<testSource>

	<directory>${project.build.directory}/generated-
test-sources/contractVerifier</directory>
				<includes>

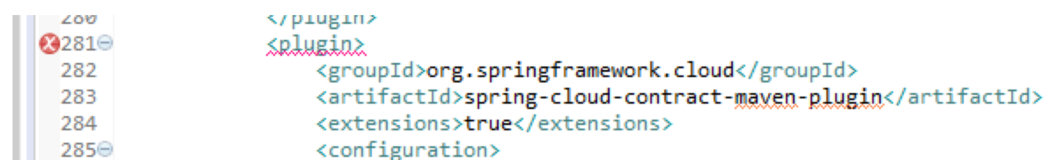
	<include>**/*.groovy</include>
				</includes>
			</testSource>
		</testSources>
	</configuration>
</plugin>
```

为了确保提供方对定义的合同进行投诉，您需要调用 `mvn generateTest test`

Maven 插件常见问题

Maven 插件和 STS

如果在使用 STS 时看到以下异常



当您点击标记时，您应该看到这样的 sth

```
 plugin:1.1.0.M1:convert:default-convert:process-test-
resources)
org.apache.maven.plugin.PluginExecutionException:
Execution default-convert of goal
org.springframework.cloud:spring-
 cloud-contract-maven-plugin:1.1.0.M1:convert failed. at
org.apache.maven.plugin.DefaultBuildPluginManager.execute
Mojo(DefaultBuildPluginManager.java:145) at

org.eclipse.m2e.core.internal.embedder.MavenImpl.execute(
MavenImpl.java:331) at
org.eclipse.m2e.core.internal.embedder.MavenImpl$11.call(
MavenImpl.java:1362) at
...

org.eclipse.core.internal.jobs.Worker.run(Worker.java:55)
Caused by: java.lang.NullPointerException at

org.eclipse.m2e.core.internal.builder.plexusbuildapi.Ecli
pseIncrementalBuildContext.hasDelta(EclipseIncrementalBui
ldContext.java:53) at

org.sonatype.plexus.build.incremental.ThreadBuildContext.
hasDelta(ThreadBuildContext.java:59) at
```

为了解决这个问题，请在pom.xml中提供以下部分

```
<build>
    <pluginManagement>
        <plugins>
            <!--This plugin's configuration is used to
store Eclipse m2e settings
                only. It has no influence on the Maven build
itself. -->
            <plugin>
                <groupId>org.eclipse.m2e</groupId>
                <artifactId>lifecycle-mapping</artifactId>
                <version>1.0.0</version>
                <configuration>
                    <lifecycleMappingMetadata>
                        <pluginExecutions>
                            <pluginExecution>
                                <pluginExecutionFilter>

<groupId>org.springframework.cloud</groupId>
                                    <artifactId>spring-cloud-
contract-maven-plugin</artifactId>

<versionRange>[1.0,)</versionRange>
                                    <goals>
                                        <goal>convert</goal>
                                    </goals>
                                </pluginExecutionFilter>
                                <action>
                                    <execute />
                                </action>
                            </pluginExecution>
                        </pluginExecutions>
                    </lifecycleMappingMetadata>
                </configuration>
            </plugin>
        </plugins>
    </pluginManagement>
</build>
```

Spring Cloud Contract 消费者验证者

您实际上也可以为消费者使用 Spring Cloud Contract 验证器！您可以使用插件，以便只转换合同并生成存根。要实现这一点，您需要以与提供程序相同的方式配置 Spring Cloud Contract 验证程序插件。您需要复制存储在 `src/test/resources/contracts` 中的合同，并使用以下命令生成 WireMock json 存根：`mvn generateStubs` 命令。默认生成的 WireMock 映射存储在目录 `target/mappings` 中。您的项目应该从此生成的映射创建附加工件与分类器 `stubs`，以便轻松部署到 maven 存储库。

样品配置：

```
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-
plugin</artifactId>
    <version>${verifier-plugin.version}</version>
    <executions>
        <execution>
            <goals>
                <goal>convert</goal>
                <goal>generateStubs</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

当存在时，json 存根可用于消费者自动测试。

```
@RunWith(SpringTestRunner.class)
@SpringBootTest
@AutoConfigureStubRunner
public class LoanApplicationServiceTests {

  @Autowired
  LoanApplicationService service;

  @Test
```

```
  public void shouldSuccessfullyApplyForLoan() {
    //given:
       LoanApplication application =
                    new LoanApplication(new
Client("12345678901"), 123.123);
    //when:
       LoanApplicationResult loanApplication =
service.loanApplication(application);
    // then:
       assertThat(loanApplication.loanApplicationStatus).i
sEqualTo(LoanApplicationStatus.LOAN_APPLIED);
       assertThat(loanApplication.rejectionReason).isNull(
);
  }
}
```

`LoanApplication` 下方致电 `FraudDetection` 服务。此请求由使用 Spring Cloud Contract 验证器生成的存根配置的 WireMock 服务器进行处理。

方案

可以使用 Spring Cloud Contract 验证程序处理场景。所有您需要做的是在创建合同时坚持正确的命名约定。公约要求包括后面是下划线的订单号。

```
my_contracts_dir\
  scenario1\
    1_login.groovy
    2_showCart.groovy
    3_logout.groovy
```

这样的树将导致 Spring Cloud Contract 验证器生成名为 `scenario1` 的 WireMock 场景和三个步骤：

- 登录标记为 `Started`，指向：
- showCart 标记为 `Step1` 指向：

- 注销标记为 `Step2`，这将关闭场景。

有关 WireMock 场景的更多详细信息，请参见 http://wiremock.org/stateful-behaviour.html

Spring Cloud Contract 验证者还将生成具有保证执行顺序的测试。

存根和传递依赖

我们创建的 Maven 和 Gradle 插件是为您添加创建存根 jar 的任务。可能有问题的是，当重用存根时，您可以错误地导入所有这些存根依赖关系！即使你有几个不同的罐子，建造一个 Maven 的工件，他们都有一个 pom：

```
├── github-webhook-0.0.1.BUILD-20160903.075506-1-
stubs.jar
├── github-webhook-0.0.1.BUILD-20160903.075506-1-
stubs.jar.sha1
├── github-webhook-0.0.1.BUILD-20160903.075655-2-
stubs.jar
├── github-webhook-0.0.1.BUILD-20160903.075655-2-
stubs.jar.sha1
├── github-webhook-0.0.1.BUILD-SNAPSHOT.jar
├── github-webhook-0.0.1.BUILD-SNAPSHOT.pom
├── github-webhook-0.0.1.BUILD-SNAPSHOT-stubs.jar
├── ...
└── ...
```

使用这些依赖关系有三种可能性，以便不会对传递依赖性产生任何问题。

将所有应用程序依赖项标记为可选

如果在 `github-webhook` 应用程序中，我们将所有的依赖项标记为可选的，当您将 `github-webhook` 存根包含在另一个应用程序中（或者当依赖关系由 Stub Runner 下载）时，因为所有的依赖关系是可选的，它们不会被下载。

为存根创建一个单独的 artifactid

如果你创建一个单独的 artifactid，那么你可以设置任何你想要的方式。例如通过没有依赖关系。

排除消费者方面的依赖关系

作为消费者，如果将 stub 依赖关系添加到类路径中，则可以显式排除不需要的依赖关系。

Spring Cloud Contract 验证器消息

Spring Cloud Contract 验证器允许您验证使用消息传递作为通信方式的应用程序。我们所有的集成都使用 Spring，但您也可以自己创建并使用它。

集成

您可以使用四种集成配置之一：

- Apache Camel
- Spring Integration
- Spring Cloud Stream
- Spring AMQP

由于我们使用 Spring Boot，因此如果您已经将上述的一个库添加到类路径中，

那么将自动设置所有的消息传递配置。

重要	记住将@AutoConfigureMessageVerifier 放在生成的测试的基类上。否则 Spri 部分将无法正常工作。

手动集成测试

测试使用的主界面是 org.springframework.cloud.contract.verifier.messaging.MessageVerifier。它定义了如何发送和接收消息。您可以创建自己的实现来实现相同的目标。

在测试中，您可以注册 ContractVerifierMessageExchange 发送和接收遵循合同的消息。然后将@AutoConfigureMessageVerifier 添加到您的测试中，例如

```
@RunWith(SpringTestRunner.class)
@SpringBootTest
@AutoConfigureMessageVerifier
public static class MessagingContractTests {

  @Autowired
  private MessageVerifier verifier;
  ...
}
```

注意	如果您的测试也需要存根，则@AutoConfigureStubRunner 包括消息传递配置，

发行人端测试一代

在您的 DSL 中拥有 `input` 或 `outputMessage` 部分将导致在发布商方面创建测试。默认情况下，将创建 JUnit 测试，但是也可以创建 Spock 测试。

我们应该考虑三个主要场景：

- 情况 1：没有输入消息产生输出消息。输出消息由应用程序内部的组件触发（例如调度程序）
- 情况 2：输入消息触发输出消息
- 方案 3：输入消息被消耗，没有输出消息

情景 1（无输入讯息）

对于给定的合同：

```
def contractDsl = Contract.make {
        label 'some_label'
        input {
                triggeredBy('bookReturnedTriggered()')
        }
        outputMessage {
                sentTo('activemq:output')
                body('''{ "bookName" : "foo" }''')
                headers {
                        header('BOOK-NAME', 'foo')
                }
        }
}
```

将创建以下 JUnit 测试：

```
'''
 // when:
  bookReturnedTriggered();
```

```
 // then:
  ContractVerifierMessage response =
contractVerifierMessaging.receive("activemq:output");
  assertThat(response).isNotNull();
  assertThat(response.getHeader("BOOK-
NAME")).isEqualTo("foo");
 // and:
  DocumentContext parsedJson =
JsonPath.parse(contractVerifierObjectMapper.writeValueAsS
tring(response.getPayload()));

assertThatJson(parsedJson).field("bookName").isEqualTo("f
oo");
'''
```

并且将创建以下 Spock 测试:

```
'''
 when:
  bookReturnedTriggered()

 then:
  ContractVerifierMessage response =
contractVerifierMessaging.receive('activemq:output')
  assert response != null
  response.getHeader('BOOK-NAME')  == 'foo'
 and:
  DocumentContext parsedJson =
JsonPath.parse(contractVerifierObjectMapper.writeValueAsS
tring(response.payload))

assertThatJson(parsedJson).field("bookName").isEqualTo("f
oo")

'''
```

情景 2（输入触发输出）

对于给定的合同:

```
def contractDsl = Contract.make {
      label 'some_label'
```

```
		input {
			messageFrom('jms:input')
			messageBody([
					bookName: 'foo'
			])
			messageHeaders {
				header('sample', 'header')
			}
		}
		outputMessage {
			sentTo('jms:output')
			body([
					bookName: 'foo'
			])
			headers {
				header('BOOK-NAME', 'foo')
			}
		}
}
```

将创建以下 JUnit 测试:

```
'''
// given:
 ContractVerifierMessage inputMessage =
contractVerifierMessaging.create(
  "{\\"bookName\\":\\"foo\\"}"
, headers()
  .header("sample", "header"));

// when:
 contractVerifierMessaging.send(inputMessage,
"jms:input");

// then:
 ContractVerifierMessage response =
contractVerifierMessaging.receive("jms:output");
 assertThat(response).isNotNull();
 assertThat(response.getHeader("BOOK-
NAME")).isEqualTo("foo");
// and:
 DocumentContext parsedJson =
JsonPath.parse(contractVerifierObjectMapper.writeValueAsS
tring(response.getPayload()));
```

```
assertThatJson(parsedJson).field("bookName").isEqualTo("f
oo");
'''
```

并且将创建以下 Spock 测试:

```
"""\
given:
   ContractVerifierMessage inputMessage =
contractVerifierMessaging.create(
    '''{"bookName":"foo"}''',
    ['sample': 'header']
  )

when:
   contractVerifierMessaging.send(inputMessage,
'jms:input')

then:
   ContractVerifierMessage response =
contractVerifierMessaging.receive('jms:output')
   assert response !- null
   response.getHeader('BOOK-NAME')  == 'foo'
and:
   DocumentContext parsedJson =
JsonPath.parse(contractVerifierObjectMapper.writeValueAsS
tring(response.payload))

assertThatJson(parsedJson).field("bookName").isEqualTo("f
oo")
"""
```

情景 3（无输出讯息）

对于给定的合同:

```
def contractDsl = Contract.make {
        label 'some_label'
        input {
                messageFrom('jms:delete')
                messageBody([
```

```
                                bookName: 'foo'
                ])
                messageHeaders {
                        header('sample', 'header')
                }
                assertThat('bookWasDeleted()')
        }
}
```

将创建以下 JUnit 测试:

```
'''
// given:
 ContractVerifierMessage inputMessage =
contractVerifierMessaging.create(
        "{\\"bookName\\":\\"foo\\"}"
, headers()
        .header("sample", "header"));

// when:
 contractVerifierMessaging.send(inputMessage,
"jms:delete");

// then:
 bookWasDeleted();
'''
```

并且将创建以下 Spock 测试:

```
'''
given:
         ContractVerifierMessage inputMessage =
contractVerifierMessaging.create(
                \'\'\'{"bookName":"foo"}\'\'\',
                ['sample': 'header']
        )

when:
         contractVerifierMessaging.send(inputMessage,
'jms:delete')

then:
         noExceptionThrown()
```

```
        bookWasDeleted()
'''
```

消费者存根侧代

与 HTTP 部分不同 - 在消息传递中，我们需要使用存根发布 JAR 中的 Groovy DSL。然后在消费者端进行解析，创建适当的 stubbed 路由。

有关更多信息，请参阅 Stub Runner 消息部分。

Maven 的

```
<dependencies>
        <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-starter-stream-
rabbit</artifactId>
        </dependency>

        <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-starter-contract-
stub-runner</artifactId>
                <scope>test</scope>
        </dependency>
        <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-stream-test-
support</artifactId>
                <scope>test</scope>
        </dependency>
</dependencies>

<dependencyManagement>
        <dependencies>
                <dependency>

        <groupId>org.springframework.cloud</groupId>
                        <artifactId>spring-cloud-
dependencies</artifactId>
```

```
                        <version>Dalston.BUILD-
SNAPSHOT</version>
                        <type>pom</type>
                        <scope>import</scope>
                </dependency>
        </dependencies>
</dependencyManagement>
```

摇篮

```
ext {
        contractsDir = file("mappings")
        stubsOutputDirRoot =
file("${project.buildDir}/production/${project.name}-
stubs/")
}

// Automatically added by plugin:
// copyContracts - copies contracts to the output folder
from which JAR will be created
// verifierStubsJar - JAR with a provided stub suffix
// the presented publication is also added by the plugin
but you can modify it as you wish

publishing {
        publications {
                stubs(MavenPublication) {
                        artifactId "${project.name}-stubs"
                        artifact verifierStubsJar
                }
        }
}
```

Spring Cloud Contract Stub Runner

使用 Spring Cloud Contract 验证程序时可能遇到的一个问题是将生成的 WireMock JSON 存根从服务器端传递到客户端（或各种客户端）。在消息传递的客户端生成方面也是如此。

复制 JSON 文件/手动设置客户端进行消息传递是不成问题的。

这就是为什么我们会介绍可以为您自动下载和运行存根的 Spring Cloud Contract Stub Runner。

快照版本

将其他快照存储库添加到您的 build.gradle 以使用快照版本，每次成功构建后都会自动上传：

Maven 的

```
<repositories>
	<repository>
		<id>spring-snapshots</id>
		<name>Spring Snapshots</name>
		<url>https://repo.spring.io/snapshot</url>
		<snapshots>
			<enabled>true</enabled>
		</snapshots>
	</repository>
	<repository>
		<id>spring-milestones</id>
		<name>Spring Milestones</name>
		<url>https://repo.spring.io/milestone</url>
		<snapshots>
			<enabled>false</enabled>
		</snapshots>
	</repository>
	<repository>
		<id>spring-releases</id>
		<name>Spring Releases</name>
		<url>https://repo.spring.io/release</url>
		<snapshots>
			<enabled>false</enabled>
		</snapshots>
	</repository>
</repositories>
<pluginRepositories>
	<pluginRepository>
		<id>spring-snapshots</id>
		<name>Spring Snapshots</name>
```

最简单的方法是集中保留存根的方式。例如，您可以将它们作为 JAR 存储在

```
            <url>https://repo.spring.io/snapshot</url>
            <snapshots>
                  <enabled>true</enabled>
            </snapshots>
      </pluginRepository>
      <pluginRepository>
            <id>spring-milestones</id>
            <name>Spring Milestones</name>
            <url>https://repo.spring.io/milestone</url>
            <snapshots>
                  <enabled>false</enabled>
            </snapshots>
      </pluginRepository>
      <pluginRepository>
            <id>spring-releases</id>
            <name>Spring Releases</name>
            <url>https://repo.spring.io/release</url>
            <snapshots>
                  <enabled>false</enabled>
            </snapshots>
      </pluginRepository>
</pluginRepositories>
```

摇篮

```
buildscript {
      repositories {
            mavenCentral()
            mavenLocal()
            maven { url
"http://repo.spring.io/snapshot" }
            maven { url
"http://repo.spring.io/milestone" }
            maven { url
"http://repo.spring.io/release" }
      }
```

将存根发布为 JAR

最简单的方法是集中保留存根的方式。例如，您可以将它们作为 JAR 存储在 Maven 存储库中。

提示 | 对于 Maven 和 Gradle 来说，安装程序都是开箱即用的。但是如果你想要的话可以自

Maven 的

```
<!-- First disable the default jar setup in the
properties section-->
<!-- we don't want the verifier to do a jar for us -->
<spring.cloud.contract.verifier.skip>true</spring.cloud.c
ontract.verifier.skip>

<!-- Next add the assembly plugin to your build -->
<plugin>
	<groupId>org.apache.maven.plugins</groupId>
	<artifactId>maven-assembly-plugin</artifactId>
	<executions>
		<execution>
			<id>stub</id>
			<phase>prepare-package</phase>
			<goals>
				<goal>single</goal>
			</goals>
			<inherited>false</inherited>
			<configuration>
				<attach>true</attach>

	<descriptor>$/Users/sgibb/workspace/spring/spring-
cloud-
samples/scripts/docs/../src/assembly/stub.xml</descriptor
>
			</configuration>
		</execution>
	</executions>
</plugin>

<!-- Finally setup your assembly. Below you can find the
contents of src/main/assembly/stub.xml -->
<assembly
	xmlns="http://maven.apache.org/plugins/maven-
assembly-plugin/assembly/1.1.3"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
	xsi:schemaLocation="http://maven.apache.org/plugins
/maven-assembly-plugin/assembly/1.1.3
http://maven.apache.org/xsd/assembly-1.1.3.xsd">
```

```
	<id>stubs</id>
	<formats>
		<format>jar</format>
	</formats>
	<includeBaseDirectory>false</includeBaseDirectory>
	<fileSets>
		<fileSet>
			<directory>src/main/java</directory>
			<outputDirectory>/</outputDirectory>
			<includes>

	<include>**com/example/model/*.*</include>
			</includes>
		</fileSet>
		<fileSet>

	<directory>${project.build.directory}/classes</dire
ctory>
			<outputDirectory>/</outputDirectory>
			<includes>

	<include>**com/example/model/*.*</include>
			</includes>
		</fileSet>
		<fileSet>

	<directory>${project.build.directory}/snippets/stub
s</directory>
			<outputDirectory>META-
INF/${project.groupId}/${project.artifactId}/${project.ve
rsion}/mappings</outputDirectory>
			<includes>
				<include>**/*</include>
			</includes>
		</fileSet>
		<fileSet>

	<directory>$/Users/sgibb/workspace/spring/spring-
cloud-
samples/scripts/docs/../src/test/resources/contracts</dir
ectory>
			<outputDirectory>META-
INF/${project.groupId}/${project.artifactId}/${project.ve
rsion}/contracts</outputDirectory>
```

```
                        <includes>
                                <include>**/*.groovy</include>
                        </includes>
                </fileSet>
        </fileSets>
</assembly>
```

摇篮

```
ext {
        contractsDir = file("mappings")
        stubsOutputDirRoot =
file("${project.buildDir}/production/${project.name}-
stubs/")
}

// Automatically added by plugin:
// copyContracts - copies contracts to the output folder
from which JAR will be created
// verifierStubsJar - JAR with a provided stub suffix
// the presented publication is also added by the plugin
but you can modify it as you wish

publishing {
        publications {
                stubs(MavenPublication) {
                        artifactId "${project.name}-stubs"
                        artifact verifierStubsJar
                }
        }
}
```

模块

Stub Runner 核心

为服务合作者运行存根。作为服务合同处理存根允许使用 stub-runner 作为 Consumer Driven Contracts 的实现。

Stub Runner 允许您自动下载提供的依赖项的存根，为其启动 WireMock 服务器，并为其提供适当的存根定义。对于消息传递，定义了特殊的存根路由。

运行存根

限制

重要	StubRunner 可能会在测试之间关闭端口时出现问题。您可能会遇到您遇到端口冲突 下文，一切正常。但是当上下文不同（例如不同的存根或不同的配置文件）时，您 服务器，否则在每个测试的不同端口上运行它们。

运行使用主应用程序

您可以将以下选项设置为主类：

```
-c, --classifier                Suffix for the jar
containing stubs (e.
                               g. 'stubs' if the stub jar
would
                               have a 'stubs' classifier for
stubs:
                               foobar-stubs ). Defaults to
'stubs'
                               (default: stubs)
--maxPort, --maxp <Integer>     Maximum port value to be
assigned to
                               the WireMock instance.
Defaults to
                               15000 (default: 15000)
--minPort, --minp <Integer>     Minimum port value to be
assigned to
                               the WireMock instance.
Defaults to
                               10000 (default: 10000)
-p, --password                  Password to user when
connecting to
                               repository
--phost, --proxyHost            Proxy host to use for
repository
```

```
                             requests
--pport, --proxyPort [Integer]  Proxy port to use for
repository
                             requests
-r, --root                   Location of a Jar containing
server
                             where you keep your stubs
(e.g. http:
                             //nexus.

net/content/repositories/repository)
-s, --stubs                  Comma separated list of Ivy
                             representation of jars with
stubs.
                             Eg.
groupid:artifactid1,groupid2:
                             artifactid2:classifier
-u, --username                Username to user when
connecting to
                             repository
--wo, --workOffline            Switch to work offline.
Defaults to
                             'false'
```

HTTP 存根

存根在 JSON 文档中定义，其语法在 WireMock 文档中定义

例:

```
{
   "request": {
      "method": "GET",
      "url": "/ping"
   },
   "response": {
      "status": 200,
      "body": "pong",
      "headers": {
         "Content-Type": "text/plain"
      }
   }
```

```
}
```

查看注册的映射

每个 stubbed 协作者公开 `__/admin/`端点下定义的映射列表。

消息存根

根据提供的 Stub Runner 依赖关系和 DSL，消息路由将自动设置。

Stub Runner JUnit 规则

Stub Runner 附带一个 JUnit 规则，感谢您可以轻松地下载和运行给定组和工件 ID 的存根：

```
@ClassRule public static StubRunnerRule rule = new
StubRunnerRule()
             .repoRoot(repoRoot())

       .downloadStub("org.springframework.cloud.contract.v
erifier.stubs", "loanIssuance")

       .downloadStub("org.springframework.cloud.contract.v
erifier.stubs:fraudDetectionServer");
```

该规则执行后 Stub Runner 连接到您的 Maven 存储库，给定的依赖关系列表尝试：

- 下载它们
- 在本地缓存
- 将它们解压缩到临时文件夹

- 从提供的端口/提供的端口范围的随机端口上为每个 Maven 依赖关系启动 WireMock 服务器
- 为 WireMock 服务器提供所有具有有效 WireMock 定义的 JSON 文件

Stub Runner 使用 Eclipse Aether 机制下载 Maven 依赖关系。查看他们的文档了解更多信息。

由于 `StubRunnerRule` 实现了 `StubFinder`，它允许您找到已启动的存根：

```
package org.springframework.cloud.contract.stubrunner;

import java.net.URL;
import java.util.Collection;
import java.util.Map;

import org.springframework.cloud.contract.spec.Contract;

public interface StubFinder extends StubTrigger {
	/**
	 * For the given groupId and artifactId tries to
find the matching
	 * URL of the running stub.
	 *
	 * @param groupId - might be null. In that case a
search only via artifactId takes place
	 * @return URL of a running stub or throws
exception if not found
	 */
	URL findStubUrl(String groupId, String artifactId)
throws StubNotFoundException;

	/**
	 * For the given Ivy notation {@code
[groupId]:artifactId:[version]:[classifier]} tries to
	 * find the matching URL of the running stub. You
can also pass only {@code artifactId}.
	 *
```

```
        * @param ivyNotation - Ivy representation of the
Maven artifact
        * @return URL of a running stub or throws
exception if not found
        */
       URL findStubUrl(String ivyNotation) throws
StubNotFoundException;

       /**
        * Returns all running stubs
        */
       RunningStubs findAllRunningStubs();

       /**
        * Returns the list of Contracts
        */
       Map<StubConfiguration, Collection<Contract>>
getContracts();
}
```

Spock 测试中使用示例：

```
@ClassRule @Shared StubRunnerRule rule = new
StubRunnerRule()

       .repoRoot(StubRunnerRuleSpec.getResource("/m2repo/r
epository").toURI().toString())

       .downloadStub("org.springframework.cloud.contract.v
erifier.stubs", "loanIssuance")

       .downloadStub("org.springframework.cloud.contract.v
erifier.stubs:fraudDetectionServer")

def 'should start WireMock servers'() {
       expect: 'WireMocks are running'

       rule.findStubUrl('org.springframework.cloud.contrac
t.verifier.stubs', 'loanIssuance') != null
              rule.findStubUrl('loanIssuance') != null
              rule.findStubUrl('loanIssuance') ==
rule.findStubUrl('org.springframework.cloud.contract.veri
fier.stubs', 'loanIssuance')
```

```
        rule.findStubUrl('org.springframework.cloud.contrac
t.verifier.stubs:fraudDetectionServer') != null
        and:

        rule.findAllRunningStubs().isPresent('loanIssuance'
)

        rule.findAllRunningStubs().isPresent('org.springfra
mework.cloud.contract.verifier.stubs',
'fraudDetectionServer')

        rule.findAllRunningStubs().isPresent('org.springfra
mework.cloud.contract.verifier.stubs:fraudDetectionServer
')
        and: 'Stubs were registered'

        "${rule.findStubUrl('loanIssuance').toString()}/nam
e".toURL().text == 'loanIssuance'

        "${rule.findStubUrl('fraudDetectionServer').toStrin
g()}/name".toURL().text == 'fraudDetectionServer'
}
```

JUnit 测试中的使用示例：

```
@Test
public void should_start_wiremock_servers() throws
Exception {
        // expect: 'WireMocks are running'

        then(rule.findStubUrl("org.springframework.cloud.co
ntract.verifier.stubs", "loanIssuance")).isNotNull();

        then(rule.findStubUrl("loanIssuance")).isNotNull();

        then(rule.findStubUrl("loanIssuance")).isEqualTo(ru
le.findStubUrl("org.springframework.cloud.contract.verifi
er.stubs", "loanIssuance"));

        then(rule.findStubUrl("org.springframework.cloud.co
ntract.verifier.stubs:fraudDetectionServer")).isNotNull()
;
        // and:
```

```
        then(rule.findAllRunningStubs().isPresent("loanIssu
ance")).isTrue();

        then(rule.findAllRunningStubs().isPresent("org.spri
ngframework.cloud.contract.verifier.stubs",
"fraudDetectionServer")).isTrue();

        then(rule.findAllRunningStubs().isPresent("org.spri
ngframework.cloud.contract.verifier.stubs:fraudDetectionS
erver")).isTrue();
        // and: 'Stubs were registered'

        then(httpGet(rule.findStubUrl("loanIssuance").toStr
ing() + "/name")).isEqualTo("loanIssuance");

        then(httpGet(rule.findStubUrl("fraudDetectionServer
").toString() +
"/name")).isEqualTo("fraudDetectionServer");
}
```

有关如何应用 Stub Runner 的全局配置的更多信息，请查看 **JUnit 和 Spring** 的 **公共属性。**

Maven 设置

存根下载器为不同的本地存储库文件夹授予 Maven 设置。目前没有考虑存储库和配置文件的身份验证详细信息，因此您需要使用上述属性进行指定。

提供固定端口

您还可以在固定端口上运行您的存根。你可以通过两种不同的方法来实现。一个是在属性中传递它，另一个是通过 JUnit 规则的流畅 API。

流畅的 API

使用 `StubRunnerRule` 时，您可以添加一个存根下载，然后通过上次下载的存根的端口。

```
@ClassRule public static StubRunnerRule rule = new
StubRunnerRule()
             .repoRoot(repoRoot())

       .downloadStub("org.springframework.cloud.contract.v
erifier.stubs", "loanIssuance")
             .withPort(12345)

       .downloadStub("org.springframework.cloud.contract.v
erifier.stubs:fraudDetectionServer:12346");
```

您可以看到，对于此示例，以下测试是有效的：

```
then(rule.findStubUrl("loanIssuance")).isEqualTo(URI.crea
te("http://localhost:12345").toURL());
then(rule.findStubUrl("fraudDetectionServer")).isEqualTo(
URI.create("http://localhost:12346").toURL());
```

Stub Runner 与 Spring

设置 Stub Runner 项目的 Spring 配置。

通过在配置文件中提供存根列表，Stub Runner 自动下载并注册 WireMock 中所选择的存根。

如果要查找 stubbed 依赖关系的 URL，您可以自动连接 `StubFinder` 接口并使用其方法，如下所示：

```
@ContextConfiguration(classes = Config, loader =
SpringBootContextLoader)
@SpringBootTest(properties = ["
stubrunner.cloud.enabled=false",
             "stubrunner.camel.enabled=false",
```

```
	'foo=${stubrunner.runningstubs.fraudDetectionServer
.port}'])
@AutoConfigureStubRunner
@DirtiesContext
@ActiveProfiles("test")
class StubRunnerConfigurationSpec extends Specification {

	@Autowired StubFinder stubFinder
	@Autowired Environment environment
	@Value('${foo}') Integer foo

	@BeforeClass
	@AfterClass
	void setupProps() {

	System.clearProperty("stubrunner.repository.root")

	System.clearProperty("stubrunner.classifier")
	}

	def 'should start WireMock servers'() {
		expect: 'WireMocks are running'

	stubFinder.findStubUrl('org.springframework.cloud.c
ontract.verifier.stubs', 'loanIssuance') != null

	stubFinder.findStubUrl('loanIssuance') != null
			stubFinder.findStubUrl('loanIssuance')
==
stubFinder.findStubUrl('org.springframework.cloud.contrac
t.verifier.stubs', 'loanIssuance')
			stubFinder.findStubUrl('loanIssuance')
==
stubFinder.findStubUrl('org.springframework.cloud.contrac
t.verifier.stubs:loanIssuance')

	stubFinder.findStubUrl('org.springframework.cloud.c
ontract.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT') ==
stubFinder.findStubUrl('org.springframework.cloud.contrac
t.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT:stubs')

	stubFinder.findStubUrl('org.springframework.cloud.c
ontract.verifier.stubs:fraudDetectionServer') != null
```

```
		and:

	stubFinder.findAllRunningStubs().isPresent('loanIss
uance')

	stubFinder.findAllRunningStubs().isPresent('org.spr
ingframework.cloud.contract.verifier.stubs',
'fraudDetectionServer')

	stubFinder.findAllRunningStubs().isPresent('org.spr
ingframework.cloud.contract.verifier.stubs:fraudDetection
Server')
		and: 'Stubs were registered'

	"${stubFinder.findStubUrl('loanIssuance').toString(
)}/name".toURL().text == 'loanIssuance'

	"${stubFinder.findStubUrl('fraudDetectionServer').t
oString()}/name".toURL().text == 'fraudDetectionServer'
	}

	def 'should throw an exception when stub is not
found'() {
		when:

	stubFinder.findStubUrl('nonExistingService')
		then:
			thrown(StubNotFoundException)
		when:

	stubFinder.findStubUrl('nonExistingGroupId',
'nonExistingArtifactId')
		then:
			thrown(StubNotFoundException)
	}

	def 'should register started servers as environment
variables'() {
		expect:

	environment.getProperty("stubrunner.runningstubs.lo
anIssuance.port") != null

	stubFinder.findAllRunningStubs().getPort("loanIssua
```

```
nce") ==
(environment.getProperty("stubrunner.runningstubs.loanIss
uance.port") as Integer)
		and:

	environment.getProperty("stubrunner.runningstubs.fr
audDetectionServer.port") != null

	stubFinder.findAllRunningStubs().getPort("fraudDete
ctionServer") ==
(environment.getProperty("stubrunner.runningstubs.fraudDe
tectionServer.port") as Integer)
	}

	def 'should be able to interpolate a running stub
in the passed test property'() {
		given:
			int fraudPort =
stubFinder.findAllRunningStubs().getPort("fraudDetectionS
erver")
		expect:
			fraudPort > 0
			environment.getProperty("foo",
Integer) == fraudPort
			foo == fraudPort
	}

	@Configuration
	@EnableAutoConfiguration
	static class Config {}
}
```

对于以下配置文件:

```
stubrunner:
  repositoryRoot: classpath:m2repo/repository/
  ids:
    -
org.springframework.cloud.contract.verifier.stubs:loanIss
uance
    -
org.springframework.cloud.contract.verifier.stubs:fraudDe
tectionServer
```

```
    -
org.springframework.cloud.contract.verifier.stubs:bootSer
vice
  cloud:
    enabled: false
  camel:
    enabled: false

spring.cloud:
  consul.enabled: false
  service-registry.enabled: false
```

您也可以使用`@AutoConfigureStubRunner`内的属性代替使用属性。下面您可以通过设置注释的值来找到实现相同结果的示例。

```
@AutoConfigureStubRunner(
                ids =
["org.springframework.cloud.contract.verifier.stubs:loanI
ssuance",

        "org.springframework.cloud.contract.verifier.stubs:
fraudDetectionServer",

        "org.springframework.cloud.contract.verifier.stubs:
bootService"],
                repositoryRoot =
"classpath:m2repo/repository/")
```

Stub Runner Spring 为每个注册的 WireMock 服务器以以下方式注册环境变量。

Stub Runner ids `com.example:foo`, `com.example:bar` 的示例。

- `stubrunner.runningstubs.foo.port`
- `stubrunner.runningstubs.bar.port`

你可以在你的代码中引用它。

Stub Runner Spring Cloud

Stub Runner 可以与 Spring Cloud 整合。

对于现实生活中的例子，你可以检查

- 制作人应用程式样本
- 消费者应用程序样本

Stubbing 服务发现

`Stub Runner Spring Cloud` 的最重要的特征就是它的存在

- `DiscoveryClient`
- `Ribbon` `ServerList`

这意味着无论您是否使用 Zookeeper，Consul，Eureka 或其他任何事情，您都不需要在测试中。我们正在启动您的依赖项的 WireMock 实例，只要您直接使用 `Feign`，负载平衡 `RestTemplate` 或 `DiscoveryClient`，我们会告诉您的应用程序来调用这些 stubbed 服务器，而不是调用真实的服务发现工具。

例如这个测试将通过

```
def 'should make service discovery work'() {
        expect: 'WireMocks are running'

        "${stubFinder.findStubUrl('loanIssuance').toString()}/name".toURL().text == 'loanIssuance'

        "${stubFinder.findStubUrl('fraudDetectionServer').toString()}/name".toURL().text == 'fraudDetectionServer'
        and: 'Stubs can be reached via load service discovery'
```

```
      restTemplate.getForObject('http://loanIssuance/name', String) == 'loanIssuance'

      restTemplate.getForObject('http://someNameThatShouldMapFraudDetectionServer/name', String) ==
'fraudDetectionServer'
}
```

对于以下配置文件

```
spring.cloud:
  zookeeper.enabled: false
  consul.enabled: false
eureka.client.enabled: false
stubrunner:
  camel.enabled: false
  idsToServiceIds:
    ivyNotation: someValueInsideYourCode
    fraudDetectionServer:
someNameThatShouldMapFraudDetectionServer
```

测试配置文件和服务发现

在集成测试中，您通常不想既不调用发现服务（例如 Eureka）或调用服务器。

这就是为什么你创建一个额外的测试配置，你要禁用这些功能。

由于 `spring-cloud-commons` 实现这一点的某些限制，您可以通过下面的静态块来禁用这些属性（例如 Eureka）

```
    //Hack to work around https://github.com/spring-cloud/spring-cloud-commons/issues/156
    static {
        System.setProperty("eureka.client.enabled", "false");
        System.setProperty("spring.cloud.config.failFast", "false");
    }
```

附加配置

您可以使用 `stubrunner.idsToServiceIds:`地图将存根的 artifactId 与应用程序的名称进行匹配。提供：`stubrunner.cloud.ribbon.enabled` 等于 `false`，您可以禁用 Stub Runner Ribbon 支持。您可以通过提供 `stubrunner.cloud.enabled` 等于 `false` 来禁用 Stub Runner 支持

提示	默认情况下，所有服务发现都将被删除。这意味着不管事实如果你有一个现有的 Di 略。但是，如果要重用它，只需将 `stubrunner.cloud.delegate.enabled` 设置 `DiscoveryClient` 结果与已存在的结果合并。

Stub Runner 启动应用程序

Spring Cloud Contract 验证者 Stub Runner Boot 是一个 Spring Boot 应用程序，它暴露了 REST 端点来触发邮件标签并访问启动的 WireMock 服务器。

其中一个用例是在部署的应用程序上运行一些烟雾（端到端）测试。您可以在 Too Much Coding 博客的“Microservice 部署”文章中阅读更多信息 。

如何使用它?

只需添加

```
compile "org.springframework.cloud:spring-cloud-starter-
stub-runner"
```

用`@EnableStubRunnerServer` 注释一个课程，建一个胖子，你准备好了!

对于属性，请检查 **Stub Runner Spring** 部分。

端点

HTTP

- GET `/stubs` - 返回 `ivy:integer` 表示法中所有运行存根的列表
- GET `/stubs/{ivy}` - 返回给定的 `ivy` 符号的端口（当调用端点 `ivy` 也可以是 `artifactId`）

消息

消息传递

- GET `/triggers` - 返回 `ivy : [ label1, label2 …]` 表示法中所有正在运行的标签的列表
- POST `/triggers/{label}` - 执行 `label` 的触发器
- POST `/triggers/{ivy}/{label}` - 对于给定的 `ivy` 符号（当调用端点 `ivy` 也可以是 `artifactId`）时，执行具有 `label` 的触发器）

例

```
@ContextConfiguration(classes = StubRunnerBoot, loader =
SpringBootContextLoader)
@SpringBootTest(properties =
"spring.cloud.zookeeper.enabled=false")
@ActiveProfiles("test")
class StubRunnerBootSpec extends Specification {

	@Autowired StubRunning stubRunning

	def setup() {
		RestAssuredMockMvc.standaloneSetup(new
HttpStubsController(stubRunning),
```

```
                          new
TriggerController(stubRunning))
       }

       def 'should return a list of running stub servers
in "full ivy:port" notation'() {
              when:
                     String response =
RestAssuredMockMvc.get('/stubs').body.asString()
              then:
                     def root = new
JsonSlurper().parseText(response)

       root.'org.springframework.cloud.contract.verifier.s
tubs:bootService:0.0.1-SNAPSHOT:stubs' instanceof Integer
       }

       def 'should return a port on which a [#stubId] stub
is running'() {
              when:
                     def response =
RestAssuredMockMvc.get("/stubs/${stubId}")
              then:
                     response.statusCode == 200
                     response.body.as(Integer) > 0
              where:
                     stubId <<
['org.springframework.cloud.contract.verifier.stubs:bootS
ervice:+:stubs',

'org.springframework.cloud.contract.verifier.stubs:bootSe
rvice:0.0.1-SNAPSHOT:stubs',

'org.springframework.cloud.contract.verifier.stubs:bootSe
rvice:+',

'org.springframework.cloud.contract.verifier.stubs:bootSe
rvice',
                                    'bootService']
       }

       def 'should return 404 when missing stub was
called'() {
              when:
```

```
                    def response =
RestAssuredMockMvc.get("/stubs/a:b:c:d")
            then:
                    response.statusCode == 404
      }

      def 'should return a list of messaging labels that
can be triggered when version and classifier are
passed'() {
            when:
                    String response =
RestAssuredMockMvc.get('/triggers').body.asString()
            then:
                    def root = new
JsonSlurper().parseText(response)

      root.'org.springframework.cloud.contract.verifier.s
tubs:bootService:0.0.1-
SNAPSHOT:stubs'?.containsAll(["delete_book","return_book_
1","return_book_2"])
      }

      def 'should trigger a messaging label'() {
            given:
                    StubRunning stubRunning = Mock()
                    RestAssuredMockMvc.standaloneSetup(new
HttpStubsController(stubRunning), new
TriggerController(stubRunning))
            when:
                    def response =
RestAssuredMockMvc.post("/triggers/delete_book")
            then:
                    response.statusCode == 200
            and:
                    1 * stubRunning.trigger('delete_book')
      }

      def 'should trigger a messaging label for a stub
with [#stubId] ivy notation'() {
            given:
                    StubRunning stubRunning = Mock()
                    RestAssuredMockMvc.standaloneSetup(new
HttpStubsController(stubRunning), new
TriggerController(stubRunning))
```

```
		when:
			def response =
RestAssuredMockMvc.post("/triggers/$stubId/delete_book")
		then:
			response.statusCode == 200
		and:
			1 * stubRunning.trigger(stubId,
'delete_book')
		where:
			stubId <<
['org.springframework.cloud.contract.verifier.stubs:bootS
ervice:stubs',
'org.springframework.cloud.contract.verifier.stubs:bootSe
rvice', 'bootService']
	}

	def 'should throw exception when trigger is
missing'() {
		when:

	RestAssuredMockMvc.post("/triggers/missing_label")
		then:
			Exception e = thrown(Exception)
			e.message.contains("Exception occurred
while trying to return [missing_label] label.")
			e.message.contains("Available labels
are")

	e.message.contains("org.springframework.cloud.contr
act.verifier.stubs:loanIssuance:0.0.1-SNAPSHOT:stubs=[]")

	e.message.contains("org.springframework.cloud.contr
act.verifier.stubs:bootService:0.0.1-SNAPSHOT:stubs=")
	}

}
```

Stub Runner 启动服务发现

使用 Stub Runner Boot 的可能性之一就是将其用作“烟雾测试”的存根。这是什么意思？假设您不想将 50 个微服务部署到测试环境中，以检查您的应用程序是否

正常工作。您在构建过程中已经执行了一系列测试，但您也希望确保应用程序的打包正常。您可以做的是将应用程序部署到环境中，启动并运行一些测试，以确定它是否正常工作。我们可以将这些测试称为烟雾测试，因为他们的想法只是检查一些测试场景。

这种方法的问题是，如果您正在执行微服务，则很可能您正在使用服务发现工具。Stub Runner 引导允许您通过启动所需的存根并将其注册到服务发现工具中来解决此问题。让我们来看看一个这样一个设置的例子 Eureka。假设 Eureka 已经在运行。

```
@SpringBootApplication
@EnableStubRunnerServer
@EnableEurekaClient
@AutoConfigureStubRunner
public class StubRunnerBootEurekaExample {

	public static void main(String[] args) {

	SpringApplication.run(StubRunnerBootEurekaExample.class, args);
	}

}
```

如您所见，我们希望启动一个 Stub Runner 引导服务器 `@EnableStubRunnerServer`，启用 Eureka 客户端`@EnableEurekaClient`，并且我们想要使存根转移功能打开`@AutoConfigureStubRunner`。

现在我们假设我们要启动这个应用程序，以便自动注册存根。我们可以通过运行应用程序 `java -jar ${SYSTEM_PROPS} stub-runner-boot-eureka-example.jar`来执行此操作，其中`${SYSTEM_PROPS}`将包含以下属性列表

```
-
Dstubrunner.repositoryRoot=http://repo.spring.io/snapshot
s (1)
-Dstubrunner.cloud.stubbed.discovery.enabled=false (2)
-
Dstubrunner.ids=org.springframework.cloud.contract.verifi
er.stubs:loanIssuance,org.springframework.cloud.contract.
verifier.stubs:fraudDetectionServer,org.springframework.c
loud.contract.verifier.stubs:bootService (3)
-
Dstubrunner.idsToServiceIds.fraudDetectionServer=someName
ThatShouldMapFraudDetectionServer (4)

(1) - we tell Stub Runner where all the stubs reside
(2) - we don't want the default behaviour where the
discovery service is stubbed. That's why the stub
registration will be picked
(3) - we provide a list of stubs to download
(4) - we provide a list of artifactId to serviceId
mapping
```

这样您的部署应用程序可以通过服务发现将请求发送到启动的 WireMock 服务器。默认情况下，`application.yml` 可能会设置 1-3，因为它们不太可能改变。这样，只要您启动 Stub Runner 引导，您只能提供要下载的存根列表。

JUnit 和 Spring 的常用属性

可以使用系统属性或配置属性（对于 Spring）设置重复的某些属性。以下是他们的名称及其默认值：

物业名称	默认值	描述
stubrunner.minPort	10000	具有存根的起始 WireMock 端口的最小值
stubrunner.maxPort	15000	具有存根的起始 WireMock 端口的最小值
stubrunner.repositoryRoot		Maven repo 网址 如果空白，那么将调用本

物业名称	默认值	描述
stubrunner.classifier	stubs	stub 工件的默认分类器
stubrunner.workOffline	false	如果为 true，则不会联系任何远程存储库
stubrunner.ids		数组的常春藤符号存根下载
stubrunner.username		可选的用户名访问使用存根存储 JAR 的工
stubrunner.password		访问使用存根存储 JAR 的工具的可选密码

存根运动员短桩 ids

您可以通过 `stubrunner.ids` 系统属性提供存根下载。他们遵循以下模式：

```
groupId:artifactId:version:classifier:port
```

`version`，`classifier` 和 `port` 是可选的。

- 如果您不提供 `port`，则会选择一个随机的
- 如果您不提供 `classifier`，那么将采用默认值。（注意，你可以传递这样一个空的分类器 `groupId:artifactId:version:`）
- 如果您不提供 `version`，则将通过`+`，最新的将被下载

其中 `port` 表示 WireMock 服务器的端口。

重要	从版本 1.0.4 开始，您可以提供一系列您希望 Stub Runner 考虑的版本。您可以在这多信息。

取自 Aether 文件：

该方案接受任何形式的版本，将版本解释为数字和字母段的序列。字符'-'，'_'和'。'以及从数字到字母的转换，反之亦然分隔版本段。分隔符被视为等同物。

数字段在数学上进行比较，字母段被字典和区分大小写比较。但是，以下限定字符串被特别识别和处理：“alpha”=“a”<“beta”=“b”<“milestone”=“m”<“cr”=“rc”<“snapshot”<“final “=”ga“<”sp“。所有这些知名的限定词被认为比其他字符串更小/更老。空的段/字符串等于 0。

除了上述限定符之外，令牌“min”和“max”可以用作最终版本段，以表示具有给定前缀的最小/最大版本。例如，“1.2.min”表示 1.2 行中的最小版本，“1.2.max”表示 1.2 行中最大的版本。形式“[MN *]”的版本范围是“[MNmin，MNmax]”的缩写。

数字和字符串被认为是无法比拟的。在不同类型的版本段会相互冲突的情况下，比较将假定以前的段分别以 0 或“ga”段的形式进行填充，直到种类不一致被解决为止，例如“1-alpha”=“1.0.0-alpha “<”1.0.1-ga“=”1.0.1“。

Stub Runner 用于消息传递

Stub Runner 具有在内存中运行已发布存根的功能。它可以与开箱即用的以下框架集成

- Spring Integration
- Spring Cloud Stream
- Apache Camel
- Spring AMQP

它还提供了与市场上任何其他解决方案集成的入口点。

存根触发

要触发消息，只需使用`StubTrigger`接口即可：

```
package org.springframework.cloud.contract.stubrunner;

import java.util.Collection;
import java.util.Map;

public interface StubTrigger {

	/**
	 * Triggers an event by a given label for a given
{@code groupid:artifactid} notation. You can use only
{@code artifactId} too.
	 *
	 * Feature related to messaging.
	 *
	 * @return true - if managed to run a trigger
	 */
	boolean trigger(String ivyNotation, String
labelName);

	/**
	 * Triggers an event by a given label.
	 *
	 * Feature related to messaging.
	 *
	 * @return true - if managed to run a trigger
	 */
	boolean trigger(String labelName);

	/**
	 * Triggers all possible events.
	 *
	 * Feature related to messaging.
	 *
	 * @return true - if managed to run a trigger
	 */
	boolean trigger();

	/**
```

```
        * Returns a mapping of ivy notation of a
dependency to all the labels it has.
        *
        * Feature related to messaging.
        */
       Map<String, Collection<String>> labels();
}
```

为了方便起见，`StubFinder`接口扩展了`StubTrigger`，所以只需要在你的测试中使用一个。

`StubTrigger`提供以下选项来触发邮件：

按标签触发

```
stubFinder.trigger('return_book_1')
```

按组和人工制品 ids 触发

```
stubFinder.trigger('org.springframework.cloud.contract.verifier.stubs:camelService', 'return_book_1')
```

通过人工制品 ids 触发

```
stubFinder.trigger('camelService', 'return_book_1')
```

触发所有消息

```
stubFinder.trigger()
```

Stub Runner Camel

Spring Cloud Contract 验证器 Stub Runner 的消息传递模块为您提供了与 Apache Camel 集成的简单方法。对于提供的工件，它将自动下载存根并注册所需的路由。

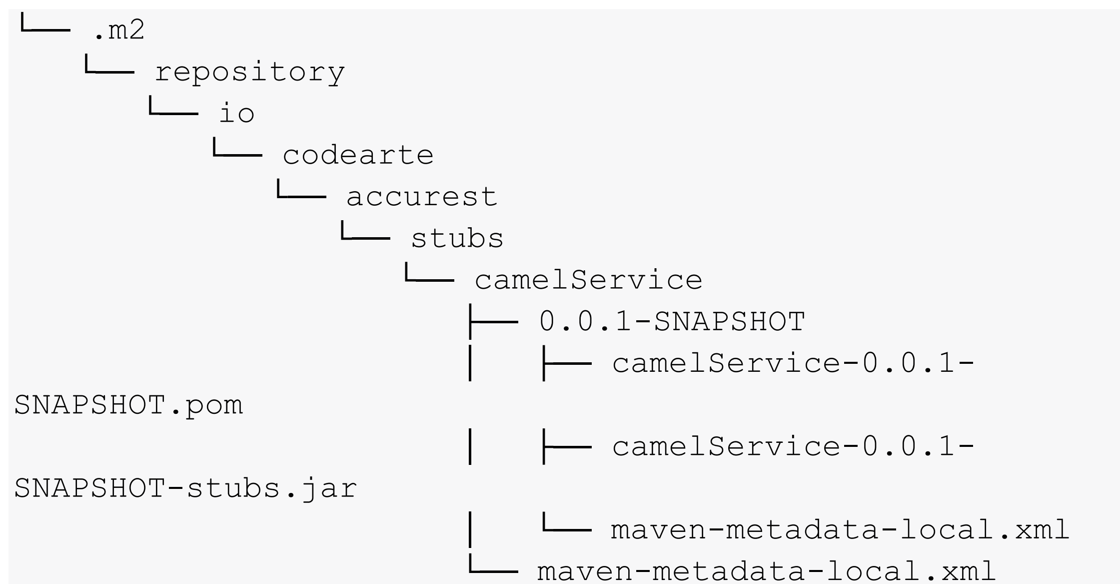
将其添加到项目中

在类路径上同时拥有 Apache Camel 和 Spring Cloud Contract Stub Runner 就足够了。记住使用`@AutoConfigureMessageVerifier`注释你的测试类。

例子

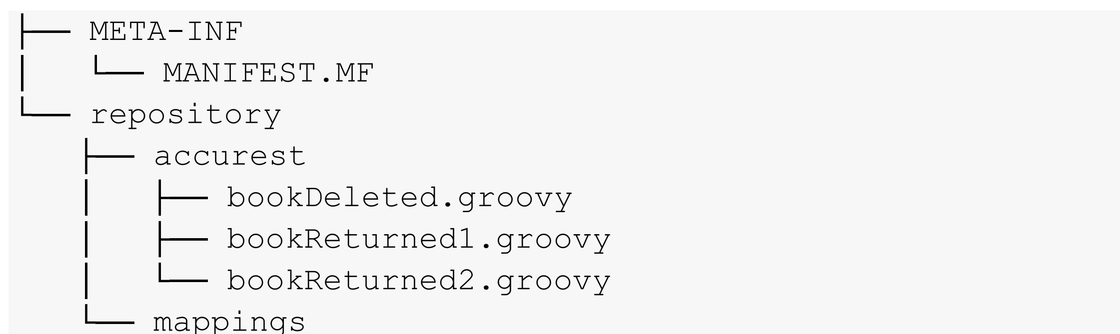
桩结构

让我们假设我们拥有以下 Maven 资源库，并为 `camelService` 应用程序配置了一个存根。



```
└── .m2
    └── repository
        └── io
            └── codearte
                └── accurest
                    └── stubs
                        └── camelService
                            ├── 0.0.1-SNAPSHOT
                            │   ├── camelService-0.0.1-SNAPSHOT.pom
                            │   ├── camelService-0.0.1-SNAPSHOT-stubs.jar
                            │   └── maven-metadata-local.xml
                            └── maven-metadata-local.xml
```

并且存根包含以下结构：



```
├── META-INF
│   └── MANIFEST.MF
└── repository
    ├── accurest
    │   ├── bookDeleted.groovy
    │   ├── bookReturned1.groovy
    │   └── bookReturned2.groovy
    └── mappings
```

让我们考虑以下合同（让我们用 **1 来表示**）：

```
Contract.make {
	label 'return_book_1'
	input {
		triggeredBy('bookReturnedTriggered()')
	}
	outputMessage {
		sentTo('jms:output')
		body('''{ "bookName" : "foo" }''')
		headers {
			header('BOOK-NAME', 'foo')
		}
	}
}
```

和 2 号

```
Contract.make {
	label 'return_book_2'
	input {
		messageFrom('jms:input')
		messageBody([
				bookName: 'foo'
		])
		messageHeaders {
			header('sample', 'header')
		}
	}
	outputMessage {
		sentTo('jms:output')
		body([
				bookName: 'foo'
		])
		headers {
			header('BOOK-NAME', 'foo')
		}
	}
}
```

情景 1（无输入讯息）

为了通过 `return_book_1` 标签触发消息，我们将使用 `StubTigger` 接口，如下所示

```
stubFinder.trigger('return_book_1')
```

接下来，我们将要收听发送到 `jms:output` 的消息的输出

```
Exchange receivedMessage =
camelContext.createConsumerTemplate().receive('jms:output
', 5000)
```

接收到的消息将通过以下断言

```
receivedMessage != null
assertThatBodyContainsBookNameFoo(receivedMessage.in.body
)
receivedMessage.in.headers.get('BOOK-NAME') == 'foo'
```

情景 2（输入触发输出）

由于路由是为您设置的，只需向 `jms:output` 目的地发送消息即可。

```
camelContext.createProducerTemplate().sendBodyAndHeaders(
'jms:input', new BookReturned('foo'), [sample: 'header'])
```

接下来我们将要收听发送到 `jms:output` 的消息的输出

```
Exchange receivedMessage =
camelContext.createConsumerTemplate().receive('jms:output
', 5000)
```

接收到的消息将通过以下断言

```
receivedMessage != null
assertThatBodyContainsBookNameFoo(receivedMessage.in.body
)
receivedMessage.in.headers.get('BOOK-NAME') == 'foo'
```

情景 3（无输出输入）

由于路由是为您设置的，只需向 `jms:output` 目的地发送消息即可。

```
camelContext.createProducerTemplate().sendBodyAndHeaders(
'jms:delete', new BookReturned('foo'), [sample:
'header'])
```

Stub Runner 整合

Spring Cloud Contract 验证器 Stub Runner 的消息传递模块为您提供了一种简单的与 Spring Integration 集成的方法。对于提供的工件，它将自动下载存根并注册所需的路由。

将其添加到项目中

在类路径上同时拥有 Apache Camel 和 Spring Cloud Contract Stub Runner 就足够了。记住使用`@AutoConfigureMessageVerifier`注释测试类。

例子

桩结构

让我们假设我们拥有以下 Maven 仓库，并为 `integrationService` 应用程序配置了一个存根。

```
└── .m2
    └── repository
        └── io
            └── codearte
                └── accurest
                    └── stubs
```

```
                    └── integrationService
                        ├── 0.0.1-SNAPSHOT
                        │   ├── integrationService-0.0.1-
SNAPSHOT.pom
                        │   ├── integrationService-0.0.1-
SNAPSHOT-stubs.jar
                        │   └── maven-metadata-local.xml
                        └── maven-metadata-local.xml
```

并且存根包含以下结构:

```
├── META-INF
│   └── MANIFEST.MF
└── repository
    ├── accurest
    │   ├── bookDeleted.groovy
    │   ├── bookReturned1.groovy
    │   └── bookReturned2.groovy
    └── mappings
```

让我们考虑以下合同（让我们用 **1 来表示**）：

```
Contract.make {
	label 'return_book_1'
	input {
		triggeredBy('bookReturnedTriggered()')
	}
	outputMessage {
		sentTo('output')
		body('''{ "bookName" : "foo" }''')
		headers {
			header('BOOK-NAME', 'foo')
		}
	}
}
```

和 **2** 号

```
Contract.make {
	label 'return_book_2'
	input {
		messageFrom('input')
```

```
		messageBody([
				bookName: 'foo'
		])
		messageHeaders {
			header('sample', 'header')
		}
	}
	outputMessage {
		sentTo('output')
		body([
				bookName: 'foo'
		])
		headers {
			header('BOOK-NAME', 'foo')
		}
	}
}
```

和以下 Spring Integration 路由:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans
xmlns="http://www.springframework.org/schema/integration"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:beans="http://www.springframework.org/schema/beans"

xsi:schemaLocation="http://www.springframework.org/schema
/beans

	http://www.springframework.org/schema/beans/spring-
beans.xsd

	http://www.springframework.org/schema/integration

	http://www.springframework.org/schema/integration/s
pring-integration.xsd">


	<!-- REQUIRED FOR TESTING -->
	<bridge input-channel="output"
			output-channel="outputTest"/>
```

```
        <channel id="outputTest">
                <queue/>
        </channel>

</beans:beans>
```

情景 1（无输入讯息）

为了通过 `return_book_1` 标签触发一条消息，我们将使用 `StubTigger` 接口，如下所示

```
stubFinder.trigger('return_book_1')
```

接下来我们将要收听发送到 `output` 的消息的输出

```
Message<?> receivedMessage =
messaging.receive('outputTest')
```

接收到的消息将通过以下断言

```
receivedMessage != null
assertJsons(receivedMessage.payload)
receivedMessage.headers.get('BOOK-NAME') == 'foo'
```

情景 2（输入触发输出）

由于路由是为您设置的，只需向 `output` 目的地发送一条消息即可。

```
messaging.send(new BookReturned('foo'), [sample:
'header'], 'input')
```

接下来，我们将要收听发送到 `output` 的消息的输出

```
Message<?> receivedMessage =
messaging.receive('outputTest')
```

接收到的消息将通过以下断言

```
receivedMessage != null
assertJsons(receivedMessage.payload)
receivedMessage.headers.get('BOOK-NAME') == 'foo'
```

情景 3（无输出输入）

由于路由是为您设置的，只需向 `input` 目的地发送消息即可。

```
messaging.send(new BookReturned('foo'), [sample:
'header'], 'delete')
```

Stub Runner 流

Spring Cloud Contract 验证器 Stub Runner 的消息传递模块为您提供了与 Spring Stream 集成的简单方式。对于提供的工件，它将自动下载存根并注册所需的路由。

警告	在 Stub Runner 与 Stream 的集成中，`messageFrom` 或 `sentTo` 字符串首先被解析 果没有这样的 `destination`，它被解析为频道名称。

将其添加到项目中

在类路径上同时拥有 Apache Camel 和 Spring Cloud Contract Stub Runner 就足够了。记住用`@AutoConfigureMessageVerifier` 注释你的测试类。

例子

桩结构

让我们假设我们拥有以下 Maven 仓库，并为 `streamService` 应用程序配置了一个存根。

```
└── .m2
    └── repository
        └── io
            └── codearte
                └── accurest
                    └── stubs
                        └── streamService
                            ├── 0.0.1-SNAPSHOT
                            │   ├── streamService-0.0.1-
SNAPSHOT.pom
                            │   ├── streamService-0.0.1-
SNAPSHOT-stubs.jar
                            │   └── maven-metadata-local.xml
                            └── maven-metadata-local.xml
```

并且存根包含以下结构:

```
├── META-INF
│   └── MANIFEST.MF
└── repository
    ├── accurest
    │   ├── bookDeleted.groovy
    │   ├── bookReturned1.groovy
    │   └── bookReturned2.groovy
    └── mappings
```

让我们考虑以下合同（让我们用 **1 来表示**）:

```
Contract.make {
      label 'return_book_1'
      input { triggeredBy('bookReturnedTriggered()') }
      outputMessage {
            sentTo('returnBook')
            body('''{ "bookName" : "foo" }''')
            headers { header('BOOK-NAME', 'foo') }
      }
}
```

和 **2** 号

```
Contract.make {
      label 'return_book_2'
```

```
        input {
                messageFrom('bookStorage')
                messageBody([
                        bookName: 'foo'
                ])
                messageHeaders { header('sample',
'header') }
        }
        outputMessage {
                sentTo('returnBook')
                body([
                        bookName: 'foo'
                ])
                headers { header('BOOK-NAME', 'foo') }
        }
}
```

和以下 Spring 配置:

```
stubrunner.repositoryRoot: classpath:m2repo/repository/
stubrunner.ids:
org.springframework.cloud.contract.verifier.stubs:streamS
ervice:0.0.1-SNAPSHOT:stubs

spring:
  cloud:
    stream:
      bindings:
        output:
          destination: returnBook
        input:
          destination: bookStorage

server:
  port: 0

debug: true
```

情景 1（无输入讯息）

为了通过 `return_book_1` 标签触发一条消息，我们将使用 `StubTrigger` 接口，如下所示

```
stubFinder.trigger('return_book_1')
```

接下来，我们将要收听发送到 `destination` 为 `returnBook` 的频道的消息的输出

```
Message<?> receivedMessage =
messaging.receive('returnBook')
```

接收到的消息将通过以下断言

```
receivedMessage != null
assertJsons(receivedMessage.payload)
receivedMessage.headers.get('BOOK-NAME') == 'foo'
```

情景 2（输入触发输出）

由于路由是为您设置的，只需向 `bookStorage` `destination` 发送消息即可。

```
messaging.send(new BookReturned('foo'), [sample:
'header'], 'bookStorage')
```

接下来我们将要收听发送到 `returnBook` 的消息的输出

```
Message<?> receivedMessage =
messaging.receive('returnBook')
```

接收到的消息将通过以下断言

```
receivedMessage != null
assertJsons(receivedMessage.payload)
receivedMessage.headers.get('BOOK-NAME') == 'foo'
```

情景 3（无输出输入）

由于路由是为您设置的，只需向 `output` 目的地发送消息即可。

```
messaging.send(new BookReturned('foo'), [sample:
'header'], 'delete')
```

Stub Runner Spring AMQP

Spring Cloud Contract 验证器 Stub Runner 的消息传递模块提供了一种简单的方法来与 Spring AMQP 的 Rabbit 模板集成。对于提供的工件，它将自动下载存根并注册所需的路由。

集成尝试独立运行，即不与运行的 RabbitMQ 消息代理交互。它期望在应用程序上下文中使用 `RabbitTemplate`，并将其用作 spring boot 测试`@SpyBean`。因此，它可以使用 mockito 间谍功能来验证和内省应用程序发送的消息。

在消费消费者方面，它考虑了所有`@RabbitListener`注释端点以及应用程序上下文中的所有“SimpleMessageListenerContainer”。

由于消息通常发送到 AMQP 中的交换机，消息合同中包含交换机名称作为目标。另一方的消息侦听器绑定到队列。绑定将交换机连接到队列。如果触发消息合约，Spring AMQP 存根转移器集成将在与该交换机匹配的应用程序上下文中查找绑定。然后它从 Spring 交换机收集队列，并尝试查找绑定到这些队列的消息侦听器。消息被触发到所有匹配的消息监听器。

将其添加到项目中

在类路径上同时拥有 Spring AMQP 和 Spring Cloud Contract Stub Runner 就足够了，并设置属性 `stubrunner.amqp.enabled=true`。记住用 `@AutoConfigureMessageVerifier` 注释你的测试类。

例子

桩结构

让我们假设我们拥有以下 Maven 资源库，并为 `spring-cloud-contract-amqp-test` 应用程序配置了一个存根。

```
└── .m2
    └── repository
        └── com
            └── example
                └── spring-cloud-contract-amqp-test
                    ├── 0.4.0-SNAPSHOT
                    │   ├── spring-cloud-contract-amqp-test-
0.4.0-SNAPSHOT.pom
                    │   ├── spring-cloud-contract-amqp-test-
0.4.0-SNAPSHOT-stubs.jar
                    │   └── maven-metadata-local.xml
                    └── maven-metadata-local.xml
```

并且存根包含以下结构：

```
├── META-INF
│   └── MANIFEST.MF
└── contracts
    └── shouldProduceValidPersonData.groovy
```

让我们考虑下列合约：

```
Contract.make {
    // Human readable description
    description 'Should produce valid person data'
```

```
    // Label by means of which the output message can be
triggered
    label 'contract-test.person.created.event'
    // input to the contract
    input {
        // the contract will be triggered by a method
        triggeredBy('createPerson()')
    }
    // output message of the contract
    outputMessage {
        // destination to which the output message will be
sent
        sentTo 'contract-test.exchange'
        headers {
            header('contentType': 'application/json')
            header('__TypeId__':
'org.springframework.cloud.contract.stubrunner.messaging.
amqp.Person')
        }
        // the body of the output message
        body ([
                id: $(consumer(9), producer(regex("[0-
9]+"))),
                name: "me"
        ])
    }
}
```

和以下 Spring 配置：

```
stubrunner:
  repositoryRoot: classpath:m2repo/repository/
  ids:
org.springframework.cloud.contract.verifier.stubs.amqp:sp
ring-cloud-contract-amqp-test:0.4.0-SNAPSHOT:stubs
  amqp:
    enabled: true
server:
  port: 0
```

触发消息

因此，为了触发使用上述合同的消息，我们将使用`StubTrigger`界面如下。

```
stubTrigger.trigger("contract-test.person.created.event")
```

消息的目的地为`contract-test.exchange`，所以 Spring AMQP 存根转移器集成查找与此交换相关的绑定。

```
@Bean
public Binding binding() {
	return BindingBuilder.bind(new
Queue("test.queue")).to(new DirectExchange("contract-
test.exchange")).with("#");
}
```

绑定定义绑定队列`test.queue`。因此，以下监听器定义是一个匹配，并使用合同消息进行调用。

```
@Bean
public SimpleMessageListenerContainer
simpleMessageListenerContainer(ConnectionFactory
connectionFactory,


	MessageListenerAdapter listenerAdapter) {
	SimpleMessageListenerContainer container = new
SimpleMessageListenerContainer();
	container.setConnectionFactory(connectionFactory);
	container.setQueueNames("test.queue");
	container.setMessageListener(listenerAdapter);

	return container;
}
```

此外，以下注释的监听器表示一个匹配并将被调用。

```
@RabbitListener(bindings = @QueueBinding(
		value = @Queue(value = "test.queue"),
		exchange = @Exchange(value = "contract-
test.exchange", ignoreDeclarationExceptions = "true")))
```

```
public void handlePerson(Person person) {
       this.person = person;
}
```

注意	该消息直接交给 `MessageListener` 与 `SimpleMessageListenerContainer` 匹

Spring AMQP 测试配置

为了避免 Spring AMQP 在测试期间尝试连接到运行的代理，我们配置了一个模拟 `ConnectionFactory`。

要禁用嘲弄的 ConnectionFactory 设置属性 `stubrunner.amqp.mockConnection=false`

```
stubrunner:
  amqp:
    mockConnection: false
```

Contract DSL

重要	请记住，在合同文件中，您必须向 `Contract` 类和 `make` 静态导入 ie `org.springframework.cloud.spec.Contract.make { … }`提供完全限 `import org.springframework.cloud.spec.Contract` 提供导入，然后调用

Contract DSL 是用 Groovy 写的，但是如果以前没有使用 Groovy，不要惊慌。语言的知识并不是真正需要的，因为我们的 DSL 只使用它的一小部分（即文字，方法调用和闭包）。DSL 还被设计为程序员可读，而不需要 DSL 本身的知识 - 它是静态类型的。

提示	Spring Cloud Contract 支持在单个文件中定义多个合同！

合同存在于 Spring Cloud Contract 验证器存储库的 `spring-cloud-contract-spec` 模块中。

我们来看一下合同定义的完整例子。

```
org.springframework.cloud.contract.spec.Contract.make {
	request {
		method 'PUT'
		url '/api/12'
		headers {
			header 'Content-Type':
'application/vnd.org.springframework.cloud.contract.verif
ier.twitter-places-analyzer.v1+json'
		}
		body '''\
		[{
			"created_at": "Sat Jul 26 09:38:57
+0000 2014",
			"id": 492967299297845248,
			"id_str": "492967299297845248",
			"text": "Gonna see you at Warsaw",
			"place":
			{
				"attributes":{},
				"bounding_box":
				{
					"coordinates":
						[[
							[-
77.119759,38.791645],
							[-
76.909393,38.791645],
							[-
76.909393,38.995548],
							[-
77.119759,38.995548]
						]],
					"type":"Polygon"
				},
				"country":"United States",
				"country_code":"US",
				"full_name":"Washington, DC",
				"id":"01fbe706f872cb32",
				"name":"Washington",
				"place_type":"city",
```

```
                            "url":
"http://api.twitter.com/1/geo/id/01fbe706f872cb32.json"
                        }
                }]
        '''
        }
        response {
                status 200
        }
}
```

不是 DSL 的所有功能都在上面的例子中使用。如果您找不到您想要的内容，请查看本页下面的段落。

您可以使用独立的 maven 命令 `mvn org.springframework.cloud:spring-cloud-contract-maven-plugin:convert` *轻松地将 Contracts 编译为 WireMock 存根映射。*

限制

警告	Spring Cloud Contract 验证器不正确支持 XML。请使用 JSON 或帮助我们实现此功
警告	对 JSON 数组的大小的验证的支持是实验性的。如果要打开它，请提供等于 `true` 的 `spring.cloud.contract.verifier.assert.size` 的值。默认情况下，此功 中提供 `assertJsonSize` 属性。
警告	由于 JSON 结构可以有任何形式，因此在 GString 中使用时使用 `value(consumer` 法正确解析它。这就是为什么我们强烈推荐使用 Groovy Map 符号。

常见的顶级元素

描述

您可以添加一个 `description` 到您的合同，除了一个任意的文本。例：

```
        org.springframework.cloud.contract.spec.Contract.ma
ke {
                        description('''
given:
        An input
when:
        Sth happens
then:
        Output
''')
                }
```

名称

您可以提供您的合同名称。假设您提供了一个名称 `should register a user`。如果这样做，则自动生成测试的名称将等于 `validate_should_register_a_user`。如果是 WireMock 存根，存根的名称也将为 `should_register_a_user.json`。

重要	请确保该名称不包含任何会使生成的测试无法编译的字符。还要记住，如果您为多 生成测试将无法编译，并且生成的存根将会相互覆盖。

忽略合同

如果您想忽略合同，您可以在插件配置中设置忽略合同的值，或者仅在合同本身设置 `ignored` 属性：

```
org.springframework.cloud.contract.spec.Contract.make {
        ignored()
}
```

HTTP 顶级元素

可以在合同定义的顶层关闭中调用以下方法。请求和响应是强制性的，优先级是可选的。

```
org.springframework.cloud.contract.spec.Contract.make {
	// Definition of HTTP request part of the contract
	// (this can be a valid request or invalid
depending
	// on type of contract being specified).
	request {
		//...
	}

	// Definition of HTTP response part of the contract
	// (a service implementing this contract should
respond
	// with following response after receiving request
	// specified in "request" part above).
	response {
		//...
	}

	// Contract priority, which can be used for
overriding
	// contracts (1 is highest). Priority is optional.
	priority 1
}
```

请求

HTTP 协议只需要在请求中指定**方法和地址**。在合同的请求定义中，相同的信息是强制性的。

```
org.springframework.cloud.contract.spec.Contract.make {
	request {
		// HTTP request method
(GET/POST/PUT/DELETE).
		method 'GET'

		// Path component of request URL is
specified as follows.
```

```
		urlPath('/users')
	}

	response {
		//...
	}
}
```

可以指定整个`url`而不是路径，但是`urlPath`是测试与**主机无关的推荐方法。**

```
org.springframework.cloud.contract.spec.Contract.make {
	request {
		method 'GET'

		// Specifying `url` and `urlPath` in one
contract is illegal.
		url('http://localhost:8888/users')
	}

	response {
		//...
	}
}
```

请求可能包含**查询参数**，这些参数在嵌套在`urlPath`或`url`的调用中的闭包中指定。

```
org.springframework.cloud.contract.spec.Contract.make {
	request {
		//...

		urlPath('/users') {

			// Each parameter is specified in form
			// `'paramName' : paramValue` where
parameter value
			// may be a simple literal or one of
matcher functions,
			// all of which are used in this
example.
			queryParameters {
```

```
                         // If a simple literal is used
as value
                         // default matcher function is
used (equalTo)
                         parameter 'limit': 100

                         // `equalTo` function simply
compares passed value
                         // using identity operator (==).
                         parameter 'filter':
equalTo("email")

                         // `containing` function matches
strings
                         // that contains passed
substring.
                         parameter 'gender':
value(consumer(containing("[mf]")), producer('mf'))

                         // `matching` function tests
parameter
                         // against passed regular
expression.
                         parameter 'offset':
value(consumer(matching("[0-9]+")), producer(123))

                         // `notMatching` functions tests
if parameter
                         // does not match passed regular
expression.
                         parameter 'loginStartsWith':
value(consumer(notMatching(".{0,2}")), producer(3))
                   }
             }

             //...
       }

       response {
             //...
       }
}
```

它可能包含其他**请求标头** ...

```
org.springframework.cloud.contract.spec.Contract.make {
	request {
		//...

		// Each header is added in form `'Header-
Name' : 'Header-Value'`.
		// there are also some helper methods
		headers {
			header 'key': 'value'
			contentType(applicationJson())
		}

		//...
	}

	response {
		//...
	}
}
```

...和**请求机构**。

```
org.springframework.cloud.contract.spec.Contract.make {
	request {
		//...

		// Currently only JSON format of request
body is supported.
		// Format will be determined from a header
or body's content.
		body '''{ "login" : "john", "name": "John
The Contract" }'''
	}

	response {
		//...
	}
}
```

响应

最小响应必须包含 **HTTP 状态代码**。

```
org.springframework.cloud.contract.spec.Contract.make {
	request {
		//...
	}
	response {
		// Status code sent by the server
		// in response to request specified above.
		status 200
	}
}
```

除了状态响应可能包含**标头**和**正文**之外，它们与请求中的方式相同（参见前一段）。

动态属性

合同可以包含一些动态属性 - 时间戳/ ids 等。您不想强制使用者将其时钟保留为始终返回相同的时间值，以便与存根匹配。这就是为什么我们允许您以两种方式在合同中提供动态部分。一个是将它们直接传递到体内，一个将它们设置在另一部分，称为 `testMatchers` 和 `stubMatchers`。

体内动态属性

您可以通过 `value` 方法设置体内的属性

```
value(consumer(...), producer(...))
value(c(...), p(...))
value(stub(...), test(...))
value(client(...), server(...))
```

或者如果您正在使用 Groovy 地图符号，您可以使用`$()`方法

```
$(consumer(...), producer(...))
$(c(...), p(...))
$(stub(...), test(...))
```

```
$(client(...), server(...))
```

所有上述方法都是相同的。这意味着 `stub` 和 `client` 方法是 `consumer` 方法的别名。我们来仔细看看我们可以在后续章节中对这些值做些什么。

正则表达式

您可以使用正则表达式在 Contract DSL 中写入请求。当您想要指出给定的响应应该被提供给遵循给定模式的请求时，这是特别有用的。此外，当您需要使用模式，而不是测试和服务器端测试时，您可以使用它。

请看下面的例子：

```
org.springframework.cloud.contract.spec.Contract.make {
	request {
		method('GET')
		url $(consumer(~/\/[0-9]{2}/),
producer('/12'))
	}
	response {
		status 200
		body(
				id: $(anyNumber()),
				surname: $(
						
	consumer('Kowalsky'),
						producer(regex('[a-
zA-Z]+'))
				),
				name: 'Jan',
				created: $(consumer('2014-02-02
12:23:43'), producer(execute('currentDate(it)'))),
				correlationId:
value(consumer('5d1f9fef-e0dc-4f3d-a7e4-72d2220dd827'),
						producer(regex('[a-
fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-
[a-fA-F0-9]{12}'))
				)
```

```
			)
			headers {
				header 'Content-Type': 'text/plain'
			}
		}
}
```

您还可以使用正则表达式仅提供通信的一方。如果这样做，那么我们将自动提供与提供的正则表达式匹配的生成的字符串。例如：

```
org.springframework.cloud.contract.spec.Contract.make {
		request {
			method 'PUT'
			url value(consumer(regex('/foo/[0-9]{5}')))
			body([
				requestElement: $(consumer(regex('[0-
9]{5}')))
			])
			headers {
				header('header',
$(consumer(regex('application\\/vnd\\.fraud\\.v1\\+json;.
*'))))
			}
		}
		response {
			status 200
			body([
				responseElement: $(producer(regex('[0-
9]{7}')))
			])
			headers {

		contentType("application/vnd.fraud.v1+json")
			}
		}
}
```

在该示例中，对于请求和响应，通信的相对侧将具有生成的相应数据。

Spring Cloud Contract 附带一系列预定义的正则表达式，您可以在合同中使用。

```
protected static final Pattern TRUE_OR_FALSE =
Pattern.compile(/(true|false)/)
protected static final Pattern ONLY_ALPHA_UNICODE =
Pattern.compile(/[\p{L}]*/)
protected static final Pattern NUMBER =
Pattern.compile('-?\\d*(\\.\\d+)?')
protected static final Pattern IP_ADDRESS =
Pattern.compile('([01]?\\d\\d?|2[0-4]\\d|25[0-
5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-
5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-
5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])')
protected static final Pattern HOSTNAME_PATTERN =
Pattern.compile('((http[s]?|ftp):\\/)\\/?([^:\\/\\s]+)(:[
0-9]{1,5})?')
protected static final Pattern EMAIL =
Pattern.compile('[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-
zA-Z]{2,4}');
protected static final Pattern URL =
Pattern.compile('((www\\.|(http|https|ftp|news|file)+\\:\
\/\\/)[_.a-z0-9-]+\\.[a-z0-
9\\/_:@=.+?,##%&~-]*[^.|\\\'|\\# |!|\\(|?|,|
|>|<|;|\\)])')
protected static final Pattern UUID =
Pattern.compile('[a-z0-9]{8}-[a-z0-9]{4}-[a-z0-9]{4}-[a-
z0-9]{4}-[a-z0-9]{12}')
protected static final Pattern ANY_DATE =
Pattern.compile('(\\d\\d\\d\\d)-(0[1-9]|1[012])-(0[1-
9]|[12][0-9]|3[01])')
protected static final Pattern ANY_DATE_TIME =
Pattern.compile('([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-
9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-
9])')
protected static final Pattern ANY_TIME =
Pattern.compile('(2[0-3]|[01][0-9]):([0-5][0-9]):([0-
5][0-9])')
protected static final Pattern NON_EMPTY =
Pattern.compile(/.+/)
protected static final Pattern NON_BLANK =
Pattern.compile(/.*(\S+|\R).*|!^\R*$/)
protected static final Pattern ISO8601_WITH_OFFSET =
Pattern.compile(/([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-
9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-
9])(\.\d{3})?(Z|[+-][01]\d:[0-5]\d)/)
```

```
protected static Pattern anyOf(String... values){
	return
Pattern.compile(values.collect({"^$it\$"}).join("|"))
}

String onlyAlphaUnicode() {
	return ONLY_ALPHA_UNICODE.pattern()
}

String number() {
	return NUMBER.pattern()
}

String anyBoolean() {
	return TRUE_OR_FALSE.pattern()
}

String ipAddress() {
	return IP_ADDRESS.pattern()
}

String hostname() {
	return HOSTNAME_PATTERN.pattern()
}

String email() {
	return EMAIL.pattern()
}

String url() {
	return URL.pattern()
}

String uuid(){
	return UUID.pattern()
}

String isoDate() {
	return ANY_DATE.pattern()
}

String isoDateTime() {
	return ANY_DATE_TIME.pattern()
}
```

```
String isoTime() {
	return ANY_TIME.pattern()
}

String iso8601WithOffset() {
	return ISO8601_WITH_OFFSET.pattern()
}

String nonEmpty() {
	return NON_EMPTY.pattern()
}

String nonBlank() {
	return NON_BLANK.pattern()
}
```

所以在你的合同中你可以这样使用它

```
Contract dslWithOptionalsInString = Contract.make {
	priority 1
	request {
		method POST()
		url '/users/password'
		headers {
			contentType(applicationJson())
		}
		body(
				email:
$(consumer(optional(regex(email()))),
producer('abc@abc.com')),
				callback_url:
$(consumer(regex(hostname())),
producer('http://partners.com'))
		)
	}
	response {
		status 404
		headers {
			contentType(applicationJson())
		}
		body(
				code: value(consumer("123123"),
producer(optional("123123"))),
```

```
                              message: "User not found by 
email = [${value(producer(regex(email())), 
consumer('not.existing@user.com'))}]"
               )
        }
}
```

传递可选参数

可以在您的合同中提供可选参数。只能有可选参数:

- *STUB* 侧请求
- 响应的 *TEST* 侧

例:

```
org.springframework.cloud.contract.spec.Contract.make {
       priority 1
       request {
               method 'POST'
               url '/users/password'
               headers {
                       contentType(applicationJson())
               }
               body(
                               email: 
$(consumer(optional(regex(email()))), 
producer('abc@abc.com')),
                               callback_url: 
$(consumer(regex(hostname())), 
producer('http://partners.com'))
               )
       }
       response {
               status 404
               headers {
                       header 'Content-Type': 
'application/json'
               }
```

```
                body(
                                code: value(consumer("123123"),
producer(optional("123123")))
                )
        }
}
```

通过使用 `optional()`方法包装身体的一部分，您实际上正在创建一个应该存在 0 次或更多次的正则表达式。

如果您选择 Spock，那么上述示例将会生成以下测试：

```
"""
 given:
  def request = given()
    .header("Content-Type", "application/json")
    .body('''{"email":"abc@abc.com","callback_url":"http:/
/partners.com"}''')

 when:
  def response = given().spec(request)
    .post("/users/password")

 then:
  response.statusCode == 404
  response.header('Content-Type')  == 'application/json'
 and:
  DocumentContext parsedJson =
JsonPath.parse(response.body.asString())

assertThatJson(parsedJson).field("code").matches("(123123
)?")
"""
```

和以下存根：

```
'''
{
  "request" : {
    "url" : "/users/password",
    "method" : "POST",
```

```
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$[?(@.email =~ /([a-zA-Z0-
9._%+-]+@[a-zA-Z0-9.-]+\\\\.[a-zA-Z]{2,4})?/)]"
    }, {
      "matchesJsonPath" : "$[?(@.callback_url =~
/((http[s]?|ftp):\\\\/)\\\\/?([^:\\\\/\\\\s]+)(:[0-
9]{1,5})?/)]"
    } ],
    "headers" : {
      "Content-Type" : {
        "equalTo" : "application/json"
      }
    }
  },
  "response" : {
    "status" : 404,
    "body" :
"{\\"code\\":\\"123123\\",\\"message\\":\\"User not found
by email == [not.existing@user.com]\\"}",
    "headers" : {
      "Content-Type" : "application/json"
    }
  },
  "priority" : 1
}
'''
```

在服务器端执行自定义方法

也可以在测试期间定义要在服务器端执行的方法调用。这样的方法可以添加到在配置中定义为“baseClassForTests”的类中。例：

合同

```
org.springframework.cloud.contract.spec.Contract.make {
      request {
            method 'PUT'
            url $(consumer(regex('^/api/[0-9]{2}$')),
producer('/api/12'))
            headers {
```

```
                   header 'Content-Type':
'application/json'
             }
             body '''\
                          [{
                                "text": "Gonna see you at
Warsaw"
                          }]
                   '''
       }
       response {
             body (
                          path: $(consumer('/api/12'),
producer(regex('^/api/[0-9]{2}$'))),
                          correlationId:
$(consumer('1223456'),
producer(execute('isProperCorrelationId($it)')))
             )
             status 200
       }
}
```

基础班

```
abstract class BaseMockMvcSpec extends Specification {

       def setup() {
             RestAssuredMockMvc.standaloneSetup(new
PairIdController())
       }

       void isProperCorrelationId(Integer correlationId) {
             assert correlationId == 123456
       }

       void isEmpty(String value) {
             assert value == null
       }

}
```

重要	您不能同时使用 String 和 `execute` 来执行连接。例如呼叫 `header('Authoriza` `execute('authToken()'))` 将导致不正确的结果。要使此工作只需调用 `header` `execute('authToken()'))`，并确保 `authToken()` 方法返回您需要的所有内容

从响应引用请求

最好的情况是提供固定值，但有时您需要在响应中引用请求。为了做到这一点，您可以从 `fromRequest()` 方法中获利，从而允许您从 HTTP 请求中引用一堆元素。您可以使用以下选项：

- `fromRequest().url()` - 返回请求 URL
- `fromRequest().query(String key)` - 返回具有给定名称的第一个查询参数
- `fromRequest().query(String key, int index)` - 返回具有给定名称的第 n 个查询参数
- `fromRequest().header(String key)` - 返回具有给定名称的第一个标题
- `fromRequest().header(String key, int index)` - 返回具有给定名称的第 n 个标题
- `fromRequest().body()` - 返回完整的请求体
- `fromRequest().body(String jsonPath)` - 从与 JSON 路径匹配的请求中返回元素

我们来看看下面的合同

}

```
Contract contractDsl = Contract.make {
	request {
		method 'GET'
		url('/api/v1/xxxx') {
			queryParameters {
				parameter("foo", "bar")
				parameter("foo", "bar2")
			}
		}
		headers {
			header(authorization(), "secret")
			header(authorization(), "secret2")
		}
		body(foo: "bar", baz: 5)
	}
	response {
		status 200
		headers {
			header(authorization(), "foo
${fromRequest().header(authorization())} bar")
		}
		body(
				url: fromRequest().url(),
				param:
fromRequest().query("foo"),
				paramIndex:
fromRequest().query("foo", 1),
				authorization:
fromRequest().header("Authorization"),
				authorization2:
fromRequest().header("Authorization", 1),
				fullBody: fromRequest().body(),
				responseFoo:
fromRequest().body('$.foo'),
				responseBaz:
fromRequest().body('$.baz'),
				responseBaz2: "Bla bla
${fromRequest().body('$.foo')} bla bla"
		)
	}
}
```

运行 JUnit 测试代码将导致创建一个或多或少这样的测试

```
// given:
 MockMvcRequestSpecification request = given()
   .header("Authorization", "secret")
   .header("Authorization", "secret2")
   .body("{\"foo\":\"bar\",\"baz\":5}");

// when:
 ResponseOptions response = given().spec(request)
   .queryParam("foo","bar")
   .queryParam("foo","bar2")
   .get("/api/v1/xxxx");

// then:
 assertThat(response.statusCode()).isEqualTo(200);

assertThat(response.header("Authorization")).isEqualTo("f
oo secret bar");
// and:
 DocumentContext parsedJson =
JsonPath.parse(response.getBody().asString());

assertThatJson(parsedJson).field("url").isEqualTo("/api/v
1/xxxx");

assertThatJson(parsedJson).field("fullBody").isEqualTo("{
\"foo\":\"bar\",\"baz\":5}");

assertThatJson(parsedJson).field("paramIndex").isEqualTo(
"bar2");

assertThatJson(parsedJson).field("responseFoo").isEqualTo
("bar");

assertThatJson(parsedJson).field("authorization2").isEqua
lTo("secret2");

assertThatJson(parsedJson).field("responseBaz").isEqualTo
(5);

assertThatJson(parsedJson).field("responseBaz2").isEqualT
o("Bla bla bar bla bla");

assertThatJson(parsedJson).field("param").isEqualTo("bar"
);
```

```
assertThatJson(parsedJson).field("authorization").isEqual
To("secret");
```

您可以看到请求中的元素在响应中已被正确引用。

生成的 WireMock 存根将看起来或多或少是这样的:

```
{
  "request" : {
    "urlPath" : "/api/v1/xxxx",
    "method" : "POST",
    "headers" : {
      "Authorization" : {
        "equalTo" : "secret2"
      }
    },
    "queryParameters" : {
      "foo" : {
        "equalTo" : "bar2"
      }
    },
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$[?(@.baz == 5)]"
    }, {
      "matchesJsonPath" : "$[?(@.foo == 'bar')]"
    } ]
  },
  "response" : {
    "status" : 200,
    "body" :
"{\"url\":\"{{{request.url}}}\",\"param\":\"{{{request.qu
ery.foo.[0]}}}\",\"paramIndex\":\"{{{request.query.foo.[1
]}}}\",\"authorization\":\"{{{request.headers.Authorizati
on.[0]}}}\",\"authorization2\":\"{{{request.headers.Autho
rization.[1]}}}\",\"fullBody\":\"{{{escapejsonbody}}}\",\
"responseFoo\":\"{{{jsonpath this
'$.foo'}}}\",\"responseBaz\":{{{jsonpath this
'$.baz'}}} ,\"responseBaz2\":\"Bla bla {{{jsonpath this
'$.foo'}}} bla bla\"}",
    "headers" : {
      "Authorization" :
"{{{request.headers.Authorization.[0]}}}"
```

```
    },
    "transformers" : [ "response-template" ]
  }
}
```

因此，发送请求作为合同 `request` 部分提出的请求将导致发送以下响应主体

```
{
  "url" : "/api/v1/xxxx?foo=bar&foo=bar2",
  "param" : "bar",
  "paramIndex" : "bar2",
  "authorization" : "secret",
  "authorization2" : "secret2",
  "fullBody" : "{\"foo\":\"bar\",\"baz\":5}",
  "responseFoo" : "bar",
  "responseBaz" : 5,
  "responseBaz2" : "Bla bla bar bla bla"
}
```

重要	此功能仅适用于版本大于或等于 2.5.1 的 WireMock。我们正在使用 WireMock 的 `re` 用 Handlebars 将 Mustache `{{{ }}}`模板转换成正确的值。另外我们正在注册 2 个 可嵌入 JSON 的格式转义请求正文。另一个是 `jsonpath` 对于给定的参数知道如何在

匹配部分的动态属性

如果您一直在使用 Pact，这似乎很熟悉。很多用户习惯于在身体和设定合约的动态部分之间进行分隔。

这就是为什么你可以从两个不同的部分获利。一个称为 `stubMatchers`，您可以在其中定义应该存在于存根中的动态值。您可以在合同的 `request` 或 `inputMessage` 部分设置。另一个称为 `testMatchers`，它存在于合同的 `response` 或 `outputMessage` 方面。

目前，我们仅支持具有以下匹配可能性的基于 JSON 路径的匹配器。对于 `stubMatchers`：

- `byEquality()` - 通过提供的 JSON 路径从响应中获取的值需要等于合同中提供的值
- `byRegex(…)` - 通过提供的 JSON 路径从响应中获取的值需要与正则表达式匹配
- `byDate()` - 通过提供的 JSON 路径从响应中获取的值需要与 ISO Date 的正则表达式匹配
- `byTimestamp()` - 通过提供的 JSON 路径从响应中获取的值需要与 ISO DateTime 的正则表达式匹配
- `byTime()` - 通过提供的 JSON 路径从响应中获取的值需要匹配 ISO 时间的正则表达式

对于 `testMatchers`：

- `byEquality()` - 通过提供的 JSON 路径从响应中获取的值需要等于合同中提供的值
- `byRegex(…)` - 通过提供的 JSON 路径从响应中获取的值需要与正则表达式匹配
- `byDate()` - 通过提供的 JSON 路径从响应中获取的值需要与 ISO Date 的正则表达式匹配
- `byTimestamp()` - 通过提供的 JSON 路径从响应中获取的值需要匹配 ISO DateTime 的正则表达式

- `byTime()` - 通过提供的 JSON 路径从响应中获取的值需要匹配 ISO 时间的正则表达式

- `byType()` - 通过提供的 JSON 路径从响应中获取的值需要与合同中的响应正文中定义的类型相同。`byType` 可以关闭，您可以设置 `minOccurrence` 和 `maxOccurrence`。这样你可以断定集合的大小。

- `byCommand(…)` - 通过提供的 JSON 路径从响应中获取的值将作为您提供的自定义方法的输入传递。例如 `byCommand('foo($it)')`将导致调用匹配 JSON 路径的值将被通过的 `foo` 方法。

我们来看看下面的例子：

```
Contract contractDsl = Contract.make {
	request {
		method 'GET'
		urlPath '/get'
		body([
				duck: 123,
				alpha: "abc",
				number: 123,
				aBoolean: true,
				date: "2017-01-01",
				dateTime: "2017-01-01T01:23:45",
				time: "01:02:34",
				valueWithoutAMatcher: "foo",
				valueWithTypeMatch: "string"
		])
		stubMatchers {
			jsonPath('$.duck', byRegex("[0-
9]{3}"))
			jsonPath('$.duck', byEquality())
			jsonPath('$.alpha',
byRegex(onlyAlphaUnicode()))
			jsonPath('$.alpha', byEquality())
			jsonPath('$.number',
byRegex(number()))
```

```
                        jsonPath('$.aBoolean',
byRegex(anyBoolean()))
                        jsonPath('$.date', byDate())
                        jsonPath('$.dateTime', byTimestamp())
                        jsonPath('$.time', byTime())
                }
                headers {
                        contentType(applicationJson())
                }
        }
        response {
                status 200
                body([
                                duck: 123,
                                alpha: "abc",
                                number: 123,
                                aBoolean: true,
                                date: "2017-01-01",
                                dateTime: "2017-01-01T01:23:45",
                                time: "01:02:34",
                                valueWithoutAMatcher: "foo",
                                valueWithTypeMatch: "string",
                                valueWithMin: [
                                        1,2,3
                                ],
                                valueWithMax: [
                                        1,2,3
                                ],
                                valueWithMinMax: [
                                        1,2,3
                                ],
                                valueWithMinEmpty: [],
                                valueWithMaxEmpty: [],
                ])
                testMatchers {
                        // asserts the jsonpath value against
manual regex
                        jsonPath('$.duck', byRegex("[0-
9]{3}"))
                        // asserts the jsonpath value against
the provided value
                        jsonPath('$.duck', byEquality())
                        // asserts the jsonpath value against
some default regex
```

```
                    jsonPath('$.alpha',
byRegex(onlyAlphaUnicode()))
                    jsonPath('$.alpha', byEquality())
                    jsonPath('$.number',
byRegex(number()))
                    jsonPath('$.aBoolean',
byRegex(anyBoolean()))
                    // asserts vs inbuilt time related
regex
                    jsonPath('$.date', byDate())
                    jsonPath('$.dateTime', byTimestamp())
                    jsonPath('$.time', byTime())
                    // asserts that the resulting type is
the same as in response body
                    jsonPath('$.valueWithTypeMatch',
byType())
                    jsonPath('$.valueWithMin', byType {
                          // results in verification of
size of array (min 1)
                          minOccurrence(1)
                    })
                    jsonPath('$.valueWithMax', byType {
                          // results in verification of
size of array (max 3)
                          maxOccurrence(3)
                    })
                    jsonPath('$.valueWithMinMax', byType {
                          // results in verification of
size of array (min 1 & max 3)
                          minOccurrence(1)
                          maxOccurrence(3)
                    })
                    jsonPath('$.valueWithMinEmpty', byType
{
                          // results in verification of
size of array (min 0)
                          minOccurrence(0)
                    })
                    jsonPath('$.valueWithMaxEmpty', byType
{
                          // results in verification of
size of array (max 0)
                          maxOccurrence(0)
                    })
```

```
                    // will execute a method
`assertThatValueIsANumber`
                    jsonPath('$.duck',
byCommand('assertThatValueIsANumber($it)'))
                }
                headers {
                    contentType(applicationJson())
                }
        }
}
```

在这个例子中，我们在匹配器部分提供合同的动态部分。对于请求部分，您可以看到对于所有字段，但是 `valueWithoutAMatcher` 我们正在明确地设置我们希望存根包含的正则表达式的值。对于 `valueWithoutAMatcher`，验证将以与不使用匹配器相同的方式进行 - 在这种情况下，测试将执行相等检查。

对于 `testMatchers` 部分的响应方面，我们以类似的方式定义所有的动态部分。唯一的区别是我们也有 `byType` 匹配器。在这种情况下，我们正在检查 4 个字段，我们正在验证测试的响应是否具有一个值，其 JSON 路径与给定字段匹配的类型与响应主体中定义的相同，

- 对于`$.valueWithTypeMatch` - 我们只是检查类型是否相同
- 对于`$.valueWithMin` - 我们正在检查类型，并声明大小是否大于或等于最小出现次数
- 对于`$.valueWithMax` - 我们正在检查类型，并声明大小是否小于或等于最大值
- 对于`$.valueWithMinMax` - 我们正在检查类型，并确定大小是否在最小和最大值之间

所得到的测试或多或少会看起来像这样（请注意，我们将自动生成的断言与匹配器与 `and` 部分分开）：

```
// given:
 MockMvcRequestSpecification request = given()
   .header("Content-Type", "application/json")
   .body("{\"duck\":123,\"alpha\":\"abc\",\"number\":123,\"aBoolean\":true,\"date\":\"2017-01-01\",\"dateTime\":\"2017-01-01T01:23:45\",\"time\":\"01:02:34\",\"valueWithoutAMatcher\":\"foo\",\"valueWithTypeMatch\":\"string\"}");

// when:
 ResponseOptions response = given().spec(request)
   .get("/get");

// then:
 assertThat(response.statusCode()).isEqualTo(200);
 assertThat(response.header("Content-Type")).matches("application/json.*");
// and:
 DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());

assertThatJson(parsedJson).field("valueWithoutAMatcher").isEqualTo("foo");
// and:
 assertThat(parsedJson.read("$.duck", String.class)).matches("[0-9]{3}");
 assertThat(parsedJson.read("$.duck", Integer.class)).isEqualTo(123);
 assertThat(parsedJson.read("$.alpha", String.class)).matches("[\\p{L}]*");
 assertThat(parsedJson.read("$.alpha", String.class)).isEqualTo("abc");
 assertThat(parsedJson.read("$.number", String.class)).matches("-?\\d*(\\.\\d+)?");
 assertThat(parsedJson.read("$.aBoolean", String.class)).matches("(true|false)");
 assertThat(parsedJson.read("$.date", String.class)).matches("(\\d\\d\\d\\d)-(0[1-9]|1[012])-(0[1-9]|[12][0-9]|3[01])");
```

```
 assertThat(parsedJson.read("$.dateTime",
String.class)).matches("([0-9]{4})-(1[0-2]|0[1-9])-
(3[01]|0[1-9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-
9]):([0-5][0-9])");
 assertThat(parsedJson.read("$.time",
String.class)).matches("(2[0-3]|[01][0-9]):([0-5][0-
9]):([0-5][0-9])");
 assertThat((Object)
parsedJson.read("$.valueWithTypeMatch")).isInstanceOf(jav
a.lang.String.class);
 assertThat((Object)
parsedJson.read("$.valueWithMin")).isInstanceOf(java.util
.List.class);
 assertThat(parsedJson.read("$.valueWithMin",
java.util.Collection.class)).hasSizeGreaterThanOrEqualTo(
1);
 assertThat((Object)
parsedJson.read("$.valueWithMax")).isInstanceOf(java.util
.List.class);
 assertThat(parsedJson.read("$.valueWithMax",
java.util.Collection.class)).hasSizeLessThanOrEqualTo(3);
 assertThat((Object)
parsedJson.read("$.valueWithMinMax")).isInstanceOf(java.u
til.List.class);
 assertThat(parsedJson.read("$.valueWithMinMax",
java.util.Collection.class)).hasSizeBetween(1, 3);
 assertThat((Object)
parsedJson.read("$.valueWithMinEmpty")).isInstanceOf(java
.util.List.class);
 assertThat(parsedJson.read("$.valueWithMinEmpty",
java.util.Collection.class)).hasSizeGreaterThanOrEqualTo(
0);
 assertThat((Object)
parsedJson.read("$.valueWithMaxEmpty")).isInstanceOf(java
.util.List.class);
 assertThat(parsedJson.read("$.valueWithMaxEmpty",
java.util.Collection.class)).hasSizeLessThanOrEqualTo(0);
 assertThatValueIsANumber(parsedJson.read("$.duck"));
```

和 WireMock 这样的 stub：

```
                                '''
{
  "request" : {
```

```
    "urlPath" : "/get",
    "method" : "GET",
    "headers" : {
      "Content-Type" : {
        "matches" : "application/json.*"
      }
    },
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$[?(@.valueWithoutAMatcher ==
'foo')]"
    }, {
      "matchesJsonPath" : "$[?(@.valueWithTypeMatch ==
'string')]"
    }, {
      "matchesJsonPath" :
"$.list.some.nested[?(@.anothervalue == 4)]"
    }, {
      "matchesJsonPath" :
"$.list.someother.nested[?(@.anothervalue == 4)]"
    }, {
      "matchesJsonPath" :
"$.list.someother.nested[?(@.json == 'with value')]"
    }, {
      "matchesJsonPath" : "$[?(@.duck =~ /([0-9]{3})/)]"
    }, {
      "matchesJsonPath" : "$[?(@.duck == 123)]"
    }, {
      "matchesJsonPath" : "$[?(@.alpha =~
/([\\\\p{L}]*)/)]"
    }, {
      "matchesJsonPath" : "$[?(@.alpha == 'abc')]"
    }, {
      "matchesJsonPath" : "$[?(@.number =~
/(-?\\\\d*(\\\\.\\\\d+)?)/)]"
    }, {
      "matchesJsonPath" : "$[?(@.aBoolean =~
/((true|false))/)]"
    }, {
      "matchesJsonPath" : "$[?(@.date =~
/((\\\\d\\\\d\\\\d\\\\d)-(0[1-9]|1[012])-(0[1-9]|[12][0-
9]|3[01]))/)]"
    }, {
```

```
      "matchesJsonPath" : "$[?(@.dateTime =~ /(([0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9]))/)]"
    }, {
      "matchesJsonPath" : "$[?(@.time =~ /((2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9]))/)]"
    }, {
      "matchesJsonPath" : "$.list.some.nested[?(@.json =~ /(.*)/)]"
    } ]
  },
  "response" : {
    "status" : 200,
    "body" : "{\\"duck\\":123,\\"alpha\\":\\"abc\\",\\"number\\":123,\\"aBoolean\\":true,\\"date\\":\\"2017-01-01\\",\\"dateTime\\":\\"2017-01-01T01:23:45\\",\\"time\\":\\"01:02:34\\",\\"valueWithoutAMatcher\\":\\"foo\\",\\"valueWithTypeMatch\\":\\"string\\",\\"valueWithMin\\":[1,2,3],\\"valueWithMax\\":[1,2,3],\\"valueWithMinMax\\":[1,2,3]}",
    "headers" : {
      "Content-Type" : "application/json"
    }
  }
}
'''
```

JAX-RS 支持

我们支持 JAX-RS 2 Client API。基类需要定义 `protected WebTarget webTarget` 和服务器初始化，现在唯一的选择如何测试 JAX-RS API 是启动一个 Web 服务器。

使用身体的请求需要设置内容类型，否则将使用 `application/octet-stream`。

为了使用 JAX-RS 模式，请使用以下设置：

```
testMode === 'JAXRSCLIENT'
```

生成测试 API 的示例:

```
'''
 // when:
  Response response = webTarget
    .path("/users")
    .queryParam("limit", "10")
    .queryParam("offset", "20")
    .queryParam("filter", "email")
    .queryParam("sort", "name")
    .queryParam("search", "55")
    .queryParam("age", "99")
    .queryParam("name", "Denis.Stepanov")
    .queryParam("email", "bob@email.com")
    .request()
    .method("GET");

  String responseAsString =
response.readEntity(String.class);

 // then:
  assertThat(response.getStatus()).isEqualTo(200);
 // and:
  DocumentContext parsedJson =
JsonPath.parse(responseAsString);

assertThatJson(parsedJson).field("property1").isEqualTo("
a");
'''
```

异步支持

如果您在服务器端使用异步通信（您的控制器正在返回 `Callable`, `DeferredResult` 等等，然后在合同中您必须在 `response` 部分中提供 `async()`方法。 ：

```
org.springframework.cloud.contract.spec.Contract.make {
    request {
```

```
        method GET()
        url '/get'
    }
    response {
        status 200
        body 'Passed'
        async()
    }
}
```

使用上下文路径

Spring Cloud Contract 支持上下文路径。

重要	为了完全支持上下文路径，唯一改变的是在 PRODUCER 端的切换。自动生成测试需要

消费者方面保持不变，为了让生成的测试通过，您必须切换 **EXPLICIT** 模式。

Maven 的

```
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-
plugin</artifactId>
    <version>${spring-cloud-contract.version}</version>
    <extensions>true</extensions>
    <configuration>
        <testMode>EXPLICIT</testMode>
    </configuration>
</plugin>
```

摇篮

```
contracts {
            testMode = 'EXPLICIT'
}
```

这样就可以生成**不**使用 MockMvc 的测试。这意味着您正在生成真实的请求，您需要设置生成的测试的基类以在真正的套接字上工作。

让我们想象下面的合同：

```
org.springframework.cloud.contract.spec.Contract.make {
	request {
		method 'GET'
		url '/my-context-path/url'
	}
	response {
		status 200
	}
}
```

以下是一个如何设置基类和 Rest Assured 的示例，以使所有操作都正常工作。

```
import com.jayway.restassured.RestAssured;
import org.junit.Before;
import
org.springframework.boot.context.embedded.LocalServerPort
;
import
org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(classes =
ContextPathTestingBaseClass.class, webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
class ContextPathTestingBaseClass {

	@LocalServerPort int port;

	@Before
	public void setup() {
		RestAssured.baseURI = "http://localhost";
		RestAssured.port = this.port;
	}
}
```

这样一来：

- 您自动生成测试中的所有请求都将发送到包含上下文路径的实际端点（例如 `/my-context-path/url`）

- 您的合同反映出您具有上下文路径，因此您生成的存根也将具有该信息（例如，在存根中您将看到您也调用了`/my-context-path/url`）

消息传递顶级元素

消息传递的 DSL 与重点在 HTTP 上的 DSL 有点不同。

由方法触发的输出

可以通过调用方法来触发输出消息（例如，调度程序启动并发送消息）

```
def dsl = Contract.make {
	// Human readable description
	description 'Some description'
	// Label by means of which the output message can
be triggered
	label 'some_label'
	// input to the contract
	input {
		// the contract will be triggered by a
method
		triggeredBy('bookReturnedTriggered()')
	}
	// output message of the contract
	outputMessage {
		// destination to which the output message
will be sent
		sentTo('output')
		// the body of the output message
		body('''{ "bookName" : "foo" }''')
		// the headers of the output message
		headers {
			header('BOOK-NAME', 'foo')
		}
	}
}
```

在这种情况下，如果将执行一个称为`bookReturnedTriggered`的方法，输出消息将被发送到`output`。在消息**发布者的**一方，我们将生成一个测试，该测试将调用该方法来触发该消息。在**消费者**端，您可以使用`some_label`触发消息。

由消息触发的输出

可以通过接收消息来触发输出消息。

```
def dsl = Contract.make {
	description 'Some Description'
	label 'some_label'
	// input is a message
	input {
		// the message was received from this
destination
		messageFrom('input')
		// has the following body
		messageBody([
			bookName: 'foo'
		])
		// and the following headers
		messageHeaders {
			header('sample', 'header')
		}
	}
	outputMessage {
		sentTo('output')
		body([
			bookName: 'foo'
		])
		headers {
			header('BOOK-NAME', 'foo')
		}
	}
}
```

在这种情况下，如果 `input` 目的地收到正确的消息，则输出消息将被发送到 `output`。在消息**发布者的**一方，我们将生成一个测试，它将输入消息发送到定义的目的地。在**消费者**端，您可以向输入目的地发送消息，也可以使用 `some_label` 触发消息。

消费者/生产者

在 HTTP 中，您有一个概念 `client` / `stub and `server` / `test` 符号。您也可以在消息中使用它们，但是我们还提供了下面提供的 `consumer` 和 `produer` 方法（请注意，您可以使用`$`或 `value` 方法来提供 `consumer` 和 `producer` 部分）

```
Contract.make {
	label 'some_label'
	input {
		messageFrom value(consumer('jms:output'),
producer('jms:input'))
		messageBody([
				bookName: 'foo'
		])
		messageHeaders {
			header('sample', 'header')
		}
	}
	outputMessage {
		sentTo $(consumer('jms:input'),
producer('jms:output'))
		body([
				bookName: 'foo'
		])
	}
}
```

一个文件中的多个合同

可以在一个文件中定义多个合同。这样的合同的例子可以这样看

```
import org.springframework.cloud.contract.spec.Contract

[
        Contract.make {
            name("should post a user")
            request {
                method 'POST'
                url('/users/1')
            }
            response {
                status 200
            }
        },
        Contract.make {
            request {
                method 'POST'
                url('/users/2')
            }
            response {
                status 200
            }
        }
]
```

在这个例子中，一个合同有`name`字段，另一个没有。这将导致生成两个或多或少这样的测试：

```
package
org.springframework.cloud.contract.verifier.tests.com.hel
lo;

import com.example.TestBase;
import com.jayway.jsonpath.DocumentContext;
import com.jayway.jsonpath.JsonPath;
import
com.jayway.restassured.module.mockmvc.specification.MockM
vcRequestSpecification;
import com.jayway.restassured.response.ResponseOptions;
import org.junit.Test;
```

```
import static
com.jayway.restassured.module.mockmvc.RestAssuredMockMvc.
*;
import static
com.toomuchcoding.jsonassert.JsonAssertion.assertThatJson
;
import static org.assertj.core.api.Assertions.assertThat;

public class V1Test extends TestBase {

	@Test
	public void validate_should_post_a_user() throws
Exception {
		// given:
			MockMvcRequestSpecification request =
given();

		// when:
			ResponseOptions response =
given().spec(request)
					.post("/users/1");

		// then:

	assertThat(response.statusCode()).isEqualTo(200);
	}

	@Test
	public void validate_withList_1() throws Exception
{
		// given:
			MockMvcRequestSpecification request =
given();

		// when:
			ResponseOptions response =
given().spec(request)
					.post("/users/2");

		// then:

	assertThat(response.statusCode()).isEqualTo(200);
	}
```

```
}
```

请注意，对于具有 `name` 字段的合同，生成的测试方法名为 `validate_should_post_a_user`。对于没有名称的人 `validate_withList_1`。它对应于文件 `WithList.groovy` 的名称和列表中的合同索引。

生成的存根将看起来像这样

```
should post a user.json
1_WithList.json
```

您可以看到第一个文件从合同中获取了 `name` 参数。第二个获得了以索引为前缀的合同文件 `WithList.groovy` 的名称（在这种情况下，合同在文件中的合同列表中具有索引 `1`）。

提示	正如你可以看到，如果您的合同名称更好，那么您的测试更有意义。

定制

扩展 DSL

可以向 DSL 提供自己的功能。此功能的关键要求是保持静态兼容性。下面你可以看到一个例子：

- 创建具有可重用类的 JAR
- 在 DSL 中引用这些类

这里可以找到完整的例子。

普通 JAR

下面你可以找到我们将在 DSL 中重用的三个类。

PatternUtils 包含消费者和制作者使用的功能。

```
package com.example;

import java.util.regex.Pattern;

/**
 * If you want to use {@link Pattern} directly in your
tests
 * then you can create a class resembling this one. It
can
 * contain all the {@link Pattern} you want to use in the
DSL.
 *
 * <pre>
 * {@code
 * request {
 *     body(
 *         [ age: $(c(PatternUtils.oldEnough()))]
 *     )
 * }
 * </pre>
 *
 * Notice that we're using both {@code $()} for dynamic
values
 * and {@code c()} for the consumer side.
 *
 * @author Marcin Grzejszczak
 */
public class PatternUtils {
	public static String tooYoung() {
		return "[0-1][0-9]";
	}

	public static Pattern oldEnough() {
```

```
		return Pattern.compile("[2-9][0-9]");
	}

	public static Pattern anyName() {
		return Pattern.compile("[a-zA-Z]+");
	}

	/**
	 * Makes little sense but it's just an example ;)
	 */
	public static Pattern ok() {
		return Pattern.compile("OK");
	}
}
```

ConsumerUtils 包含由使用功能的**消费者。**

```
package com.example;

import
org.springframework.cloud.contract.spec.internal.ClientDs
lProperty;
import
org.springframework.cloud.contract.spec.internal.DslPrope
rty;

/**
 * DSL Properties passed to the DSL from the consumer's
perspective.
 * That means that on the input side {@code Request} for
HTTP
 * or {@code Input} for messaging you can have a regular
expression.
 * On the {@code Response} for HTTP or {@code Output} for
messaging
 * you have to have a concrete value.
 *
 * @author Marcin Grzejszczak
 */
public class ConsumerUtils {
	/**
	 * Consumer side property. By using the {@link
ClientDslProperty}
```

```
        * you can omit most of boilerplate code from the
perspective
        * of dynamic values. Example
        *
        * <pre>
        * {@code
        * request {
        *    body(
        *        [ age: $(ConsumerUtils.oldEnough())]
        *    )
        * }
        * </pre>
        *
        * That way the consumer side value of age field
will be
        * a regular expression and the producer side will
be generated.
        *
        * @author Marcin Grzejszczak
        */
       public static ClientDslProperty oldEnough() {
             return new
ClientDslProperty(PatternUtils.oldEnough());
       }

       /**
        * Consumer side property. By using the {@link
ClientDslProperty}
        * you can omit most of boilerplate code from the
perspective
        * of dynamic values. Example
        *
        * <pre>
        * {@code
        * request {
        *    body(
        *        [ name: $(ConsumerUtils.anyName())]
        *    )
        * }
        * </pre>
        *
        * That way the consumer will be a regular
expression and the
```

```
	 * producer side value will be equal to {@code
marcin}
	 */
	public static DslProperty anyName() {
		return new
DslProperty<>(PatternUtils.anyName(), "marcin");
	}
}
```

ProducerUtils 包含由使用的功能**制片人。**

```
package com.example;

import
org.springframework.cloud.contract.spec.internal.ServerDs
lProperty;

/**
 * DSL Properties passed to the DSL from the producer's
perspective.
 * That means that on the input side {@code Request} for
HTTP
 * or {@code Input} for messaging you have to have a
concrete value.
 * On the {@code Response} for HTTP or {@code Output} for
messaging
 * you can have a regular expression.
 *
 * @author Marcin Grzejszczak
 */
public class ProducerUtils {

	/**
	 * Producer side property. By using the {@link
ProducerUtils}
	 * you can omit most of boilerplate code from the
perspective
	 * of dynamic values. Example
	 *
	 * <pre>
	 * {@code
	 * response {
	 *    body(
	 *       [ status: $(ProducerUtils.ok())]
```

```
         *     )
         * }
         * </pre>
         *
         * That way the producer side value of age field
will be
         * a regular expression and the consumer side will
be generated.
         */
        public static ServerDslProperty ok() {
                return new
ServerDslProperty(PatternUtils.ok());
        }
}
```

将依赖项添加到项目中

为了使插件和 IDE 能够引用常见的 JAR 类，您需要将依赖关系传递给您的项目。

测试依赖项目的依赖关系

首先将常见的 jar 依赖项添加为测试依赖关系。这样，由于您的合同文件在测试资源路径中可用，所以公用的 jar 类将自动显示在您的 Groovy 文件中。

Maven 的

```
<dependency>
        <groupId>com.example</groupId>
        <artifactId>beer-common</artifactId>
        <version>${project.version}</version>
        <scope>test</scope>
</dependency>
```

摇篮

```
testCompile("com.example:beer-common:0.0.1-SNAPSHOT")
```

测试插件依赖关系

现在你必须添加插件的依赖关系，以便在运行时重用。

Maven 的

```
<plugin>
	<groupId>org.springframework.cloud</groupId>
	<artifactId>spring-cloud-contract-maven-
plugin</artifactId>
	<version>${spring-cloud-contract.version}</version>
	<extensions>true</extensions>
	<configuration>

	<packageWithBaseClasses>com.example</packageWithBas
eClasses>
	</configuration>
	<dependencies>
		<dependency>

	<groupId>org.springframework.cloud</groupId>
			<artifactId>spring-cloud-contract-
verifier</artifactId>
			<version>${spring-cloud-
contract.version}</version>
		</dependency>
		<dependency>
			<groupId>com.example</groupId>
			<artifactId>beer-common</artifactId>
			<version>${project.version}</version>
			<scope>compile</scope>
		</dependency>
	</dependencies>
</plugin>
```

摇篮

```
classpath "com.example:beer-common:0.0.1-SNAPSHOT"
```

在 DSL 中引用类

现在您可以参考 DSL 中的课程。例:

```
package contracts.beer.rest

import org.springframework.cloud.contract.spec.Contract

import static com.example.ConsumerUtils.oldEnough
```

```
import static com.example.ProducerUtils.ok

Contract.make {
	request {
		description("""
Represents a successful scenario of getting a beer

given:
	client is old enough
when:
	he applies for a beer
then:
	we'll grant him the beer
""")
		method 'POST'
		url '/check'
		body(
				age: $(oldEnough())
		)
		headers {
			contentType(applicationJson())
		}
	}
	response {
		status 200
		body("""
			{
				"status": "${value(ok())}"
			}
			""")
		headers {
			contentType(applicationJson())
		}
	}
}
```

可插拔架构

在某些情况下，您将合同定义为其他格式，如 YAML，RAML 或 PACT。另一方面，您希望从测试和存根生成中获利。添加自己的任何一个实现是很容易的。此

外，您还可以自定义测试生成的方式（例如，您可以为其他语言生成测试），并且可以对存根生成执行相同操作（可为其他存根 http 服务器实现生成存根）。

定制合同转换器

我们假设您的合同是用 YAML 文件写成的：

```
request:
  url: /foo
  method: PUT
  headers:
    foo: bar
  body:
    foo: bar
response:
  status: 200
  headers:
    foo2: bar
  body:
    foo2: bar
```

感谢界面

```
package org.springframework.cloud.contract.spec

/**
 * Converter to be used to convert FROM {@link File} TO
{@link Contract}
 * and from {@link Contract} to {@code T}
 *
 * @param <T> - type to which we want to convert the
contract
 *
 * @author Marcin Grzejszczak
 * @since 1.1.0
 */
interface ContractConverter<T> {

	/**
```

```
	 * Should this file be accepted by the converter.
Can use the file extension
	 * to check if the conversion is possible.
	 *
	 * @param file - file to be considered for
conversion
	 * @return - {@code true} if the given
implementation can convert the file
	 */
	boolean isAccepted(File file)

	/**
	 * Converts the given {@link File} to its {@link
Contract} representation
	 *
	 * @param file - file to convert
	 * @return - {@link Contract} representation of the
file
	 */
	Collection<Contract> convertFrom(File file)

	/**
	 * Converts the given {@link Contract} to a {@link
T} representation
	 *
	 * @param contract - the parsed contract
	 * @return - {@link T} the type to which we do the
conversion
	 */
	T convertTo(Collection<Contract> contract)
}
```

您可以注册自己的合同结构转换器的实现。您的实现需要说明开始转换的条件。

此外，您必须定义如何以两种方式执行转换。

重要	创建实施后，您必须创建一个`/META-INF/spring.factories`文件，您可以在其

`spring.factories`文件的示例

```
# Converters
```

```
org.springframework.cloud.contract.spec.ContractConverter
=\
org.springframework.cloud.contract.verifier.converter.Yam
lContractConverter
```

和 YAML 实现

```
package
org.springframework.cloud.contract.verifier.converter

import groovy.transform.CompileStatic
import org.springframework.cloud.contract.spec.Contract
import
org.springframework.cloud.contract.spec.ContractConverter
import
org.springframework.cloud.contract.spec.internal.Headers
import org.yaml.snakeyaml.Yaml

/**
 * Simple converter from and to a {@link YamlContract} to
a collection of {@link Contract}
 */
@CompileStatic
class YamlContractConverter implements
ContractConverter<List<YamlContract>> {

	@Override
	public boolean isAccepted(File file) {
		String name = file.getName()
		return name.endsWith(".yml") ||
name.endsWith(".yaml")
	}

	@Override
	public Collection<Contract> convertFrom(File file)
{
		try {
			YamlContract yamlContract = new
Yaml().loadAs(new FileInputStream(file),
YamlContract.class)
			return [Contract.make {
				request {

	method(yamlContract?.request?.method)
```

```
url(yamlContract?.request?.url)
                            headers {

        yamlContract?.request?.headers?.each { String key,
Object value ->
                                            header(key,
value)
                                    }
                            }

        body(yamlContract?.request?.body)
                        }
                        response {

        status(yamlContract?.response?.status)
                            headers {

        yamlContract?.response?.headers?.each { String key,
Object value ->
                                            header(key,
value)
                                    }
                            }

        body(yamlContract?.response?.body)
                        }
                    }]
            }
            catch (FileNotFoundException e) {
                    throw new IllegalStateException(e)
            }
    }

    @Override
    public List<YamlContract>
convertTo(Collection<Contract> contracts) {
            return contracts.collect { Contract contract
->
                    YamlContract yamlContract = new
YamlContract()
                    yamlContract.request.with {
                            method =
contract?.request?.method?.clientValue
```

```
                              url =
contract?.request?.url?.clientValue
                              headers =
(contract?.request?.headers as Headers)?.asStubSideMap()
                              body =
contract?.request?.body?.clientValue as Map
                        }
                        yamlContract.response.with {
                              status =
contract?.response?.status?.clientValue as Integer
                              headers =
(contract?.response?.headers as Headers)?.asStubSideMap()
                              body =
contract?.response?.body?.clientValue as Map
                        }
                        return yamlContract
                  }
            }
}
```

契约转换器

Spring Cloud Contract 提供了协议代表合同的开箱即用支持。换句话说，而不是使用 Groovy DSL，您可以使用 Pact 文件。在本节中，我们将介绍如何为您的项目添加此类支持。

契约契约

我们将在下面的一个契约契约的例子中工作。我们将此文件放在 `src/test/resources/contracts` 文件夹下。

```
{
  "provider": {
    "name": "Provider"
  },
  "consumer": {
    "name": "Consumer"
  },
```

```
  "interactions": [
    {
      "description": "",
      "request": {
        "method": "PUT",
        "path": "/fraudcheck",
        "headers": {
          "Content-Type": "application/vnd.fraud.v1+json"
        },
        "body": {
          "clientId": "1234567890",
          "loanAmount": 99999
        },
        "matchingRules": {
          "$.body.clientId": {
            "match": "regex",
            "regex": "[0-9]{10}"
          }
        }
      },
      "response": {
        "status": 200,
        "headers": {
          "Content-Type":
"application/vnd.fraud.v1+json;charset=UTF-8"
        },
        "body": {
          "fraudCheckStatus": "FRAUD",
          "rejectionReason": "Amount too high"
        },
        "matchingRules": {
          "$.body.fraudCheckStatus": {
            "match": "regex",
            "regex": "FRAUD"
          }
        }
      }
    }
  ],
  "metadata": {
    "pact-specification": {
      "version": "2.0.0"
    },
    "pact-jvm": {
```

```
    "version": "2.4.18"
   }
  }
}
```

生产者契约

在生产者方面，您可以添加两个附加依赖关系的插件配置。一个是 Spring Cloud Contract Pact 支持，另一个表示您正在使用的当前 Pact 版本。

Maven 的

```
<plugin>
	<groupId>org.springframework.cloud</groupId>
	<artifactId>spring-cloud-contract-maven-
plugin</artifactId>
	<version>${spring-cloud-contract.version}</version>
	<extensions>true</extensions>
	<configuration>

	<packageWithBaseClasses>com.example.fraud</packageW
ithBaseClasses>
	</configuration>
	<dependencies>
		<dependency>

	<groupId>org.springframework.cloud</groupId>
			<artifactId>spring-cloud-contract-
spec-pact</artifactId>
			<version>${spring-cloud-
contract.version}</version>
		</dependency>
		<dependency>
			<groupId>au.com.dius</groupId>
			<artifactId>pact-jvm-
model</artifactId>
			<version>2.4.18</version>
		</dependency>
	</dependencies>
</plugin>
```

摇篮

```
classpath "org.springframework.cloud:spring-cloud-contract-spec-pact:${findProperty('verifierVersion') ?: verifierVersion}"
classpath 'au.com.dius:pact-jvm-model:2.4.18'
```

当您执行应用程序的构建时，将会产生一个或多或少的这样的测试

```
@Test
public void validate_shouldMarkClientAsFraud() throws Exception {
	// given:
		MockMvcRequestSpecification request = given()
				.header("Content-Type", "application/vnd.fraud.v1+json")

	.body("{\"clientId\":\"1234567890\",\"loanAmount\":99999}");

	// when:
		ResponseOptions response = given().spec(request)
				.put("/fraudcheck");

	// then:

	assertThat(response.statusCode()).isEqualTo(200);
		assertThat(response.header("Content-Type")).isEqualTo("application/vnd.fraud.v1+json;charset=UTF-8");
	// and:
		DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());

	assertThatJson(parsedJson).field("rejectionReason").isEqualTo("Amount too high");
	// and:

	assertThat(parsedJson.read("$.fraudCheckStatus", String.class)).matches("FRAUD");
}
```

并且这样的存根看起来像这样

```
{
  "uuid" : "996ae5ae-6834-4db6-8fac-358ca187ab62",
  "request" : {
    "url" : "/fraudcheck",
    "method" : "PUT",
    "headers" : {
      "Content-Type" : {
        "equalTo" : "application/vnd.fraud.v1+json"
      }
    },
    "bodyPatterns" : [ {
      "matchesJsonPath" : "$[?(@.loanAmount == 99999)]"
    }, {
      "matchesJsonPath" : "$[?(@.clientId =~ /([0-
9]{10})/)]"
    } ]
  },
  "response" : {
    "status" : 200,
    "body" :
"{\"fraudCheckStatus\":\"FRAUD\",\"rejectionReason\":\"Am
ount too high\"}",
    "headers" : {
      "Content-Type" :
"application/vnd.fraud.v1+json;charset=UTF-8"
    }
  }
}
```

消费者契约

在生产者方面，您可以添加项目依赖关系两个附加依赖关系。一个是 Spring Cloud Contract Pact 支持，另一个表示您正在使用的当前 Pact 版本。

Maven 的

```
<dependency>
	<groupId>org.springframework.cloud</groupId>
	<artifactId>spring-cloud-contract-spec-
pact</artifactId>
	<scope>test</scope>
</dependency>
```

```
<dependency>
       <groupId>au.com.dius</groupId>
       <artifactId>pact-jvm-model</artifactId>
       <version>2.4.18</version>
       <scope>test</scope>
</dependency>
```

摇篮

```
testCompile "org.springframework.cloud:spring-cloud-
contract-spec-pact"
testCompile 'au.com.dius:pact-jvm-model:2.4.18'
```

定制测试发生器

如果您想为 Java 生成不同语言的测试，或者您不满意我们为您建立 Java 测试的方式，那么您可以注册自己的实现来做到这一点。

感谢界面

```
package
org.springframework.cloud.contract.verifier.builder

import
org.springframework.cloud.contract.verifier.config.Contra
ctVerifierConfigProperties
import
org.springframework.cloud.contract.verifier.file.Contract
Metadata
/**
 * Builds a single test.
 *
 * @since 1.1.0
 */
interface SingleTestGenerator {

       /**
        * Creates contents of a single test class in which
all test scenarios from
        * the contract metadata should be placed.
        *
```

```
	 * @param properties - properties passed to the
plugin
	 * @param listOfFiles - list of parsed contracts
with additional metadata
	 * @param className - the name of the generated
test class
	 * @param classPackage - the name of the package in
which the test class should be stored
	 * @param includedDirectoryRelativePath - relative
path to the included directory
	 * @return contents of a single test class
	 */
	String buildClass(ContractVerifierConfigProperties
properties, Collection<ContractMetadata> listOfFiles,
			String className, String
classPackage, String includedDirectoryRelativePath)

	/**
	 * Extension that should be appended to the
generated test class. E.g. {@code .java} or {@code .php}
	 *
	 * @param properties - properties passed to the
plugin
	 */
	String
fileExtension(ContractVerifierConfigProperties
properties)
}
```

您可以注册自己的生成测试的实现。再次提供一个合适的 `spring.factories` 文件就足够了。例：

```
org.springframework.cloud.contract.verifier.builder.Singl
eTestGenerator=/
com.example.MyGenerator
```

自定义存根发生器

如果要为 WireMock 生成其他存根服务器的存根，就可以插入您自己的此接口的实现：

```
package
org.springframework.cloud.contract.verifier.converter

import groovy.transform.CompileStatic
import org.springframework.cloud.contract.spec.Contract
import
org.springframework.cloud.contract.verifier.file.Contract
Metadata

/**
 * Converts contracts into their stub representation.
 *
 * @since 1.1.0
 */
@CompileStatic
interface StubGenerator {

	/**
	 * Returns {@code true} if the converter can handle
the file to convert it into a stub.
	 */
	boolean canHandleFileName(String fileName)

	/**
	 * Returns the collection of converted contracts
into stubs. One contract can
	 * result in multiple stubs.
	 */
	Map<Contract, String> convertContents(String
rootName, ContractMetadata content)

	/**
	 * Returns the name of the converted stub file. If
you have multiple contracts
	 * in a single file then a prefix will be added to
the generated file. If you
	 * provide the {@link Contract#name} field then
that field will override the
	 * generated file name.
	 *
	 * Example: name of file with 2 contracts is {@code
foo.groovy}, it will be
	 * converted by the implementation to {@code
foo.json}. The recursive file
```

```
        * converter will create two files {@code
0_foo.json} and {@code 1_foo.json}
        */
       String generateOutputFileNameForInput(String
inputFileName)
}
```

您可以注册自己的生成存根的实现。再次提供一个合适的 `spring.factories` 文件就足够了。例:

```
# Stub converters
org.springframework.cloud.contract.verifier.converter.Stu
bGenerator=\
org.springframework.cloud.contract.verifier.wiremock.DslT
oWireMockClientConverter
```

默认实现是 WireMock 存根生成。

提示	您可以提供多个存根生成器实现。这样,例如从单个 DSL 作为输入,您可以例如生成

自定义 Stub Runner

如果您决定使用自定义存根生成器,则还需要使用不同的存根提供程序来运行存根的自定义方式。

让我们假设您正在使用 Moco 来构建您的存根。你写了一个正确的存根生成器,你的存根被放在一个 JAR 文件中。

为了 Stub Runner 知道如何运行存根,您必须定义一个自定义的 HTTP Stub 服务器实现。它可以看起来像这样:

```
package
org.springframework.cloud.contract.stubrunner.provider.mo
co
```

```
import com.github.dreamhead.moco.bootstrap.arg.HttpArgs
import com.github.dreamhead.moco.runner.JsonRunner
import
org.springframework.cloud.contract.stubrunner.HttpServerS
tub
import org.springframework.util.SocketUtils

class MocoHttpServerStub implements HttpServerStub {

	private boolean started
	private JsonRunner runner
	private int port

	@Override
	int port() {
		if (!isRunning()) {
			return -1
		}
		return port
	}

	@Override
	boolean isRunning() {
		return started
	}

	@Override
	HttpServerStub start() {
		return
start(SocketUtils.findAvailableTcpPort())
	}

	@Override
	HttpServerStub start(int port) {
		this.port = port
		return this
	}

	@Override
	HttpServerStub stop() {
		if (!isRunning()) {
			return this
		}
```

```
                this.runner.stop()
                return this
        }

        @Override
        HttpServerStub registerMappings(Collection<File>
stubFiles) {
                List<InputStream> streams =
stubFiles.collect { it.newInputStream() }
                this.runner =
JsonRunner.newJsonRunnerWithStreams(streams,

        HttpArgs.httpArgs().withPort(this.port).build())
                this.runner.run()
                this.started = true
                return this
        }

        @Override
        boolean isAccepted(File file) {
                return file.name.endsWith(".json")
        }
}
```

并将其注册到您的 `spring.factories` 文件中

```
# Example of a custom HTTP Server Stub
org.springframework.cloud.contract.stubrunner.HttpServerS
tub=\
org.springframework.cloud.contract.stubrunner.provider.mo
co.MocoHttpServerStub
```

这样你就可以使用 Moco 来运行存根。

重要	如果您不提供任何实现，那么将选择默认的 - 基于 WireMock 的。如果您提供多个，

自定义存根下载器

您可以自定义存根的下载方式。如果您不想以默认方式从 Nexus / Artifactory 下载 JAR，您可以设置自己的实现。下面您可以找到一个 Stub Downloader Provider 示例，它从 classpath 的测试资源获取`json`文件，将它们复制到临时文件，然后将该临时文件夹作为存根的根传递。

```
package
org.springframework.cloud.contract.stubrunner.provider.mo
co

import
org.springframework.cloud.contract.stubrunner.StubConfigu
ration
import
org.springframework.cloud.contract.stubrunner.StubDownloa
der
import
org.springframework.cloud.contract.stubrunner.StubDownloa
derBuilder
import
org.springframework.cloud.contract.stubrunner.StubRunnerO
ptions
import org.springframework.core.io.DefaultResourceLoader
import org.springframework.core.io.Resource
import
org.springframework.core.io.support.PathMatchingResourceP
atternResolver

import java.nio.file.Files

/**
 * Poor man's version of taking stubs from classpath. It
needs much more
 * love and attention to go to the main sources.
 *
 * @author Marcin Grzejszczak
 */
class ClasspathStubProvider implements
StubDownloaderBuilder {

      private static final int TEMP_DIR_ATTEMPTS = 10000
```

```
	@Override
	public StubDownloader build(StubRunnerOptions
stubRunnerOptions) {
		final StubConfiguration configuration =
stubRunnerOptions.getDependencies().first()
		PathMatchingResourcePatternResolver resolver
= new PathMatchingResourcePatternResolver(
				new DefaultResourceLoader())
		try {
			String rootFolder =
repoRoot(stubRunnerOptions) ?: "**/" +
separatedArtifact(configuration) + "/**/*.json"
			Resource[] resources =
resolver.getResources(rootFolder)
			final File tmp = createTempDir()
			tmp.deleteOnExit()
			// you'd have to write an impl to
maintain the folder structure
			// this is just for demo
			resources.each { Resource resource ->

	Files.copy(resource.getInputStream(), new File(tmp,
resource.getFile().getName()).toPath())
			}
			return new StubDownloader() {
				@Override
				public
Map.Entry<StubConfiguration, File>
downloadAndUnpackStubJar(
						StubConfiguration
stubConfiguration) {
					return new
AbstractMap.SimpleEntry(configuration, tmp)
				}
			}
		} catch (IOException e) {
			throw new IllegalStateException(e)
		}
	}

	private String repoRoot(StubRunnerOptions
stubRunnerOptions) {
```

```
		switch
(stubRunnerOptions.stubRepositoryRoot) {
			case { !it }:
				return ""
			case { String root ->
root.endsWith("**/*.json") }:
				return
stubRunnerOptions.stubRepositoryRoot
			default:
				return
stubRunnerOptions.stubRepositoryRoot + "/**/*.json"
		}
	}

	private String separatedArtifact(StubConfiguration
configuration) {
		return
configuration.getGroupId().replace(".", File.separator) +
				File.separator +
configuration.getArtifactId()
	}

	// Taken from Guava
	private File createTempDir() {
		File baseDir = new
File(System.getProperty("java.io.tmpdir"))
		String baseName = System.currentTimeMillis()
+ "-"
		for (int counter = 0; counter <
TEMP_DIR_ATTEMPTS; counter++) {
			File tempDir = new File(baseDir,
baseName + counter)
			if (tempDir.mkdir()) {
				return tempDir
			}
		}
		throw new IllegalStateException(
				"Failed to create directory
within " + TEMP_DIR_ATTEMPTS + " attempts (tried " +
baseName + "0 to " + baseName + (
						TEMP_DIR_ATTEMPTS -
1) + ")")
	}
}
```

并将其注册到您的 `spring.factories` 文件中

```
# Example of a custom Stub Downloader Provider
org.springframework.cloud.contract.stubrunner.StubDownloa
derBuilder=\
org.springframework.cloud.contract.stubrunner.provider.mo
co.ClasspathStubProvider
```

这样你就可以选择一个文件夹与你的存根的来源。

重要	如果您没有提供任何实现，那么将选择从远程备份中下载存根的默认 Aether。如果 选中。

链接

在这里，您可以找到有关 Spring Cloud Contract 验证器的有趣链接：

- Spring Cloud Contract Github Repository
- Spring Cloud Contract 样本
- Spring Cloud Contract 文档
- Accurest 遗产文件
- Spring Cloud Contract Stub Runner 文档
- Spring Cloud Contract Stub Runner 消息传递文档
- Spring Cloud Contract Gitter
- Spring Cloud Contract Maven 插件